Universidade do Minho
Escola de Engenharia

Cláudio Filipe Belo da Silva Lourenço

A Bounded Model Checker for SPARK Programs

Outubro de 2013

Universidade do Minho
Escola de Engenharia

Cláudio Filipe Belo da Silva Lourenço

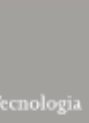
A Bounded Model Checker for SPARK Programs

Dissertação de Mestrado
Área de Especialização Engenharia Informática

Trabalho efectuado sob a orientação de
Professor Doutor Jorge Sousa Pinto
Professor Doutor Maria João Frade

Outubro de 2013

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundção para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020486, and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PTDC/EIA-CCO/105034/2008



Acknowledgments

First of all I would like to thank my supervisors, Prof. Jorge Sousa Pinto and Prof. Maria João Frade, for the great job they did supervising my thesis. I especially would like to thank them for the weekly meetings where they guided me through the right path and motivated me to work on the next week's tasks.

I am very grateful to the people from the laboratories for the good environment created and the great discussions about unsolvable problems. Especially I want to thank Iago and Jorge for being always available to help me with all kind of technical issues. To Iago I am also grateful for the suggestions he gave me for this dissertation.

To all my friends, I would like to thank for all the relaxing moments helping to get my mind out of work and allowing me to have a fresh mind when working on my thesis.

Finally, but most importantly I want to thank my family. To my parents for the education they gave me and all the sacrifices they had to make in order to do that. To my sister, who had to deal with my sometimes bad mood. I also want to mention my godfather Arménio, who always taught me how to overcome challenges and work hard to get the things I want in life.

Abstract

Formal verification of software has been an active topic in the area of computer science. Several techniques to verify software are now available, and many tools have been created over the years for different languages and using different techniques. However, for SPARK, a programming language broadly used in critical systems, only deductive verification tools based on contracts are available. The main downside of this approach is the lack of a full automation.

In this dissertation we propose an automated verification tool for SPARK code, thus contributing to fill the gap identified above. Our tool bases on an alternative technique, called *bounded model checking*, that sacrifices completeness in exchange for automation. Through grounding our work in the highly popular and successful CBMC tool for verification of C code, we investigate how to perform bounded model checking of SPARK programs, and, in particular, we present our implementation of a bounded model checker for SPARK programs called SPARK-BMC.

Experiments performed with our tool show that automatic verification of SPARK programs is feasible and useful, even though is not complete. As far as we know, there is no tool based on such an automated technique for SPARK. The tool is freely available and based on open-source technologies.

Resumo

A verificação formal de software tem sido um tópico bastante ativo na área das ciências da computação. Várias técnicas podem ser aplicadas para verificar software e ao longo dos anos surgiram várias ferramentas para diferentes linguagens usando diferentes técnicas. Para a linguagem de programação SPARK, que é especialmente usada em sistemas críticos, existem ferramentas dedutivas baseadas em contratos. Porém, esta técnica de verificação tem uma desvantagem: fraca automação.

Nesta dissertação, propomos uma ferramenta de verificação automática para código SPARK, assim contribuindo para suprir a limitação antes referida. Esta ferramenta baseia-se numa técnica alternativa denominada por 'bounded model checking' que embora não sendo completa permite automação. Baseando o nosso trabalho na popular e bem sucedida ferramenta CBMC para a verificação de código C, estudamos como levar a cabo o 'bounded model checking' de programas SPARK e, em particular apresentamos a respectiva implementação que designamos por SPARK-BMC.

As experiências que levamos a cabo com a nossa ferramenta mostram que a verificação automática de programas SPARK, ainda que não seja completa, é praticável e útil. Pelo que nos é dado a conhecer, não há nenhuma ferramenta baseada numa tal técnica automatizada para programas SPARK. A ferramenta é de acesso livre e baseia-se em tecnologias 'open-source'.

Contents

List of Figures

Acronyms

AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CFG	Control Flow Graph
CIL	C Intermediate Language
CNF	Conditional Normal Form
LTG	Labeled Transition Graph
SAT	Satisfiability
SMT	Satisfiability Modulo Theories
SA	Single Assignment
SSA	Static Single Assignment
DSA	Dynamic Single Assignment
VC	Verification Condition

Chapter 1

Introduction

This thesis fits into the area of formal verification of programs, more precisely the use of *bounded model checking* for the verification of programs written in SPARK [Bar03]. SPARK is a programming language designed for the development of high-assurance software. The development platform of SPARK programs provides a set of verification tools that allow developers to give evidence for correctness of the source code being written, and detect problems early in the software life-cycle. The existing tools for verification of SPARK programs are mainly commercialized by Altran[1] and AdaCore[2], and comprise both automatic and interactive proof for the generated verification conditions. The tools can check the absence of runtime errors, as well as functional correctness based on contracts. Lately, independent tools have been proposed, such as HOL-SPARK [Ber11] which provides interactive proof construction using the Isabelle [NWP02] proof assistant. All these tools are based on deductive verification, which implies that someone has to write the contracts and loop invariants that will be used to generate verification conditions. Since this is in general far from straightforward, and moreover the use of interactive proof tools has a steep learning curve, scalability is compromised.

An alternative approach designated by *software model checking* reaches automation in exchange for completeness or correction. In the first case,

[1] `http://www.altran.co.uk`
[2] `http://www.adacore.com`

abstraction techniques are used, taking the risk of introducing false positives. In the second case, the model checking of software just takes into account the executions with length up to a given limit, saying nothing about longer executions. Such limit is defined by the user, and verification conditions may be generated to verify if longer executions can possibly occur in the execution of the program. This approach, named *Bounded Model Checking (BMC)*, is completely automatic and precise, which in many cases allows bugs to be detected in a very efficient way. We propose an implementation of such a technique for the verification of SPARK programs. It is not our intention to replace the existing tools; instead, we intend to complement those with a new tool that (unlike the existing tools) can be used in a completely automatic way and still be useful.

We use the rest of this chapter to provide a more detailed description of the topics mentioned above. We start by presenting the general ideas about verification of programs, focusing later on the topic of this thesis: *bounded model checking of software*. A brief history and overview of the SPARK language is also given. This chapter ends with a summary of the main contributions and organization of this document.

1.1 Program Verification

Program verification consists in verifying the correctness of programs according to a specification. Let us start by reviewing some fundamental concepts related to program verification. When it is said that a program is correct, it is meant that the program is correct accordingly to a specification. This specification defines properties of the program under analysis, and is normally written as annotations (not mandatory). Such a specification expresses safety and/or liveness properties, as well as functional properties. While safety properties denote that nothing bad will happen (e.g., an assertion violation or null pointer dereference), liveness properties denote that something good will eventually happen (e.g. a program will terminate) [BBF+01]. Functional properties specify the input/output behavior of the program.

To check if a program is correct according to a specification, Verification

Conditions (VCs) are generated using the program instructions and the specification, and those must be proved by a logic solver. A verification technique based on the generation of VCs is said to be *sound* if, whenever the generated VCs are valid, the program is correct, that is, the operational behavior of the program respects the specification. Symmetrically, a verification technique is said to be *complete* if whenever a program is correct, the generated VCs are valid. Figure 1.1 shows exactly the distinction between the concepts of soundness and completeness. $VCs(P)$ represents the VCs generated for the program with respect to a given (annotated) specification P. $\models \mathsf{VCs}(P)$ denotes the validity of the VCs generated from the program and specification P. $\mathsf{CORRECT}\,(P)$ means that the program P is correct according to its specification.

Figure 1.1: Soundness vs completeness

Tools for verifying software are mainly divided in two families: tools based on a *deductive* approach and tools based on *model checking*. While *deductive verification* based on the use of a program logic and the design-by-contract principle gives full guarantees and allows for expressing properties using a rich behavior specification language [HLL+12], it is not *automatic*: it is the user's responsibility to provide *contracts* and other information required for verification to proceed, such as *loop invariants*. Such information requires a lot of effort from the user since it is often difficult to write.

The second family of techniques is based on *model checking* [CES09], more precisely *model checking of software*. This approach typically allows only for simpler properties, such as safety properties expressed as assertions in the code, but is fully automated. The fundamental idea is to create a model from the source program, and then, given a property, check if it holds in that model. However, such an approach has a main downside: *state space explosion*. This problem is common to all applications of model checking,

but the presence of data makes it worse in the case of software. This has led to the development of specific techniques for model checking of software, which we discuss later in this chapter.

1.2 Model Checking and Bounded Model Checking

The general idea of *model checking* is to check the validity of a system model accordingly to a specification [CGP99, CES09]. The model represents in general the behavior of a finite state concurrent system, and the specification, defined trough properties, represents expected behaviors of the system. The model and the specification are defined based on mathematical tools from automata theory and logic, and the verification process consists in algorithmically verifying if the model satisfies all the properties. If some property fails, a counter-example is extracted from the model.

Model checking in its best known form was initially introduced in 1981 by Clarke and Emerson [CE81]. Theirs and Sifakis' pioneering work in model checking was recognized in 2007 by a Turing Award. The formulation of the model checking technique is as follows: given a finite state transition system M, an initial state s, and a property expressed through a temporal logic formula f the technique consists in verifying if $M, s \vDash f$ holds.

The first implementations of model checkers used *explicit representation* of the state transition graph, and the exploration of states was done by traversing the graph using efficient techniques. The problem of such an approach is that the number of states grows exponentially with the number of variables, leading to the famous *state space explosion problem* [CGP99]. In the early nineteens a technique called *symbolic model checking* was proposed [Mcm92, BCM+92]. The idea is to represent the set of states symbolically, instead of representing them explicitly. Due to the success of Binary Decision Diagrams (BDDs) to symbolically represent the set of states, sometimes the expression 'symbolic model checking' is used to refer to the particular implementation of symbolic model checking based on BDDs. To have an overall

idea of symbolic model checking and BDDs see [CGP99, Mcm92].

Later in the nineteens a new technique called *Bounded Model Checking* was proposed [BCC+99]. It consists in considering only a subset of the model, more precisely the set of states requiring less than K steps to be reached (only paths with length up to K are considered). If no counter-example is found for the given bound K, nothing can be said about longer paths. However, it is possible to increment the bound until a counter-example is found or the problem becomes intractable. One big advantage of BMC is the possibility to efficiently reduce the problem to a satisfiability problem, which can be later solved by a Satisfiability (SAT) [GKSS08] or Satisfiability Modulo Theories (SMT) [dMB11] solver. Modern SAT solvers can handle propositional formulas with (more than) hundreds of thousands of variables, which makes them very suitable for this kind of problems. On the other hand, SMT solvers capture richer theories that may be useful for solving specific problems, as will be explained later.

1.3 Software Model Checking

Since its introduction model checking has been mainly used to check hardware descriptions; more recently research has been made to apply it to software systems. In the field of software the main idea is to extract a model directly from the source code, formulate on that model safety and/or liveness properties about the program execution, and then apply a model checking algorithm [BHMW09, JM09, IYG+08]. One way of modeling a program is to have in each state of the model an evaluation of the program counter, the values of all variables, and the configuration of the heap and stack. The program flow is described by transitions from one state to another.

Since software model checking is just a particular application of model checking, the techniques to represent and verify the model are those described in the previous section. However, the state space explosion is an even bigger concern here, since the state space grows exponentially with respect to many parameters, as for example the number of variables. It can be even infinite in the presence of function calls and/or concurrency [SKW08].

1.3.1 Software Model Checking using Abstraction Techniques

Abstraction can be used to reduce the impact of the state-space problem, by trading precision for efficiency [CES09]. Certain properties depend only on some parts of the program. The technique consists in constructing a model containing only the relevant parts of the program to the property of interest. Thus, the abstract model captures some, but not necessarily all the information about an execution, keeping always the control flow. There are various abstraction techniques, with *predicate abstraction* [BHMW09] the most popular. The idea is to keep a set of boolean predicates over the data, whose truth values change with the concrete program steps. The result program is called a *boolean program*, since all variables are of type *boolean*. Due to the abstraction, there may be reachable states in the abstract model that are not reached in the concrete model, so when checking for safety properties, each time a counter-example is found in the abstract model, it has to be checked if this is actually a counter-example in the original model. If it is not, this leads to the *predicate refinement* techniques, which consist in refining the predicate each time a spurious counter-example is found.

We remark that if the property happens to be valid in the abstract model, so is it in the concrete model (the use of abstraction techniques is sound).

1.3.2 Bounded Model Checking of Software

The other alternative to overcome the state space explosion problem is the use of *bounded model checking of software*. This technique only checks executions with length up to a fixed (user-provided) bound, sacrificing soundness. In software, the main idea of BMC is to encode *bounded behaviors* of the program that enjoys some given property as a logical formula whose models, if any, describe executions leading to a violation of the property. The properties to be established are assertions on the program state, included in the program through the use of *assert statements*. For every execution of the program, whenever a statement `assert` ϕ is met, the assertion ϕ must be satisfied by the current state, otherwise we say that the state violates the assertion ϕ.

Such assertions may be seen at the operational level as follows:

$$\texttt{assert}\ \phi \equiv \texttt{if}\ \neg\phi\ \texttt{then ABORT;}$$

This highlights the fact that properties expressed through assertions are indeed safety properties, in the sense that a correct program does not enter an abort state. The verification technique assumes that a satisfiability-based tool is used to find models corresponding to property violations.

At the heart of a BMC tool stands an algorithm that extracts a logical formula directly from the source code (including properties expressed as assertions), without user intervention. The algorithm begins with some preliminary simplification steps that may include the removal of side effects, or normalization into a subset of the target programming language. The next step is where information is lost, in the sense that only bounded behaviors are preserved. Given an entry-point provided by the user, the program is expanded by unwinding loops a fixed number of times, and inlining routine calls (in the presence of recursion, a bound is also applied on the length of this expansion). A program consisting of multiple routines is thus transformed into a monolithic program, which is both recursion-free and iteration-free.

To enforce soundness of BMC, an *unwinding assertion* can be placed at the end of each expanded segment of code. If the unwinding assertion (negating the condition of the loop) is not violated by any execution, then checking the transformed (bounded) code is sound. Unwinding assertions can be omitted, in which case one must always bear in mind the unsoundness of the approach.

In order to extract a logical formula from this monolithic program, it has to be transformed into a form in which the values of the variables do not change once they have been used (so that they can be seen as logical variables). This is done by converting the program into a Single Assignment (SA) form in which multiple indexed versions of each variable are used – a new version is introduced for each assignment to the original variable, so that in every execution path, once a variable has been read or assigned it will not be assigned again.

The next step is to normalize this program into *conditional normal form* (CNF): a sequence of single-branch conditional statements of the form `if` b `then` S, where S is an *atomic statement*, i.e. either an assignment, or *assert* instruction. The idea is to flatten the branching structure of the program, so that every atomic statement is guarded by the path condition leading to it.

At this point, two sets $\mathcal{C}$ and $\mathcal{P}$ can be extracted from the program: $\mathcal{C}$ includes a formula $b \rightarrow x = e$ for every statement `if` b `then` $x := e$; $\mathcal{P}$ includes a formula $b \rightarrow \phi$ for every statement `if` b `then assert` ϕ. $\mathcal{C}$ captures logically the operational contents of the program, and $\mathcal{P}$ contains the properties to be established.

If no assert statement fails in any execution of the program, one has that $\bigwedge \mathcal{P}$ is a logical consequence of $\mathcal{C}$. This can be determined by checking the satisfiability of the set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$. Any model found for it corresponds to an execution path that leads to an assertion violation. Of course, satisfiability checking is restricted to models that capture the properties of the data structures manipulated by the program, and that are specified by some background theory $\mathcal{T}$ (usually a combination of several theories). Therefore 'satisfiability' should in fact be understood as $\mathcal{T}$-satisfiability. It can be checked by a Satisfiability (SAT) or a Satisfiability Modulo Theory (SMT) solver: for our purposes the main difference is in the way numeric values and arrays are modelled.

1.4 SPARK

The SPARK programming language is basically a subset of Ada, whose first release dates from 1983 (Ada 83). Ada was created for the United States of America Department of Defense, when the department realized that their software was written in many different languages. The cost of supporting updates and maintainability for this software was very high. So, a set of requirements for a universal language meeting their requirements was created, taking into account opinions from various software experts in the government, industry and academia. When analyzing the existing languages, they realized that none of the existing programming languages could satisfy their

requirements. The solution found was to create a competition for companies to create a language meeting these requirements. The main requirements for the language included: readability, efficiency, provability and expressiveness. The contest winner came from a French team led by Jean Ichbiah, and a few years late Ada 83 was released[3]. Along the years a few more versions were released, but always keeping the initial requirements. The current main characteristics of Ada are[4]:

- **Strongly typed** - With strong typing (also called safe typing) many errors can be detected at compile time.
- **Modular** - The modularity in Ada is achieved through *packages*. The packages are divided into *specification* and *body*. The package specification defines an interface for external usage. The package body gives the implementation details of the package.
- **Object oriented** - Ada also supports Objected Oriented Programming.
- **Concurrent** - It is a built-in feature in Ada through Ada tasks. The tasks run concurrently and Ada provides mechanisms for them to communicate with each other.

Although the use of Ada can avoid many errors that are common in other languages, there are systems which require more than that. For some systems the effect of an error can lead to loss of lives or catastrophic disasters. The Ada language was chosen as a foundation for some research that has been carried out since 1970, from which SPARK was born in 1988 [Bar03].

As was said before, SPARK is a subset of Ada with some features added through annotations which are actually comments in Ada. Thus, a SPARK program can be compiled by an Ada compiler, avoiding the need for a special compiler. SPARK removes all the features that are dangerous or difficult to verify in the development of safety critical systems, such as recursion,

[3] `http://www.adaic.org/learn/materials/intro/part1/#history`
[4] `http://www.adacore.com/knowledge/technical-papers/safe-secure/`

dynamic memory allocations, access types (pointers), dynamic dispatching, and generics. Furthermore, with the annotations added in SPARK, it is possible to check the absence of errors, as well as to formally verify the program according to a specification.

The annotations can be divided in two parts: *core annotations* and *proof annotations*. The core annotations are divided into flow analysis and visibility control. Flow analysis annotations specify how the data information flows in a *subprogram*, or in other words, it specifies how each variable is used and, if assigned, it specifies its dependencies. The visibility control annotations specify what is visible from within a *subprogram* or *package*, e.g., global variables usage or *packages* importation. The proof annotations are used to specify *pre-conditions* and *post-conditions* as well as loop invariants and other assertions. These annotations are then used by the SPARK tool *Examiner* to check the validity of the core annotations, and to generate verification conditions for the proof annotations. In what follows we explore a simple SPARK example which consists only in a subprogram that swaps the values of two variables given as (in out) parameters.

```
package SP is
   procedure swap(A: in out Integer; B: in out Integer);
   --# derives A from B &
   --#          B from A;
   --# post A=B~ and B=A~;
end SP;
```

Figure 1.2: SPARK program specification

Figure 1.2 presents a SPARK package specification `SP`. It defines the procedure `swap` which has two input and output parameters. In the annotations it is specified that the output value of `A` derives from `B` and the output value of `B` derives from `A`. In the post-condition (also included as an annotation) it is stated that the output value of `A` is equal to the input value of `B`, and the other way around.

Figure 1.3 shows the body of the package `SP`. The function defines an auxiliary variable `Aux` which is used as a temporary variable. In the subprogram implementation the swap action is performed in the expected way.

```
package body SP is
   procedure swap(A: in out Integer; B: in out Integer)
   is
     Aux: Integer;
   begin
      Aux := A;
      A := B;
      B := Aux;
  end swap;
end SP;
```

Figure 1.3: SPARK program body

1.5 Contributions and Document Outline

The most important contribution of this thesis is the SPARK-BMC tool, which is capable of performing *bounded model checking* of SPARK programs. Although it is not our idea to compete with the existing available SPARK tools, we show that automatic verification of SPARK code is feasible and useful.

Chapter 2 describes some aspects of the SPARK language that are relevant in the development of a *bounded model checker*. We start by describing the motivation for a language like SPARK, and later we describe the organization of SPARK programs, the type system, relevant statements and annotations. Moreover, we present the SPARK toolset and show some examples where SPARK technology has been applied.

Chapter 3 presents the state of the art in BMC of software. The first part of that chapter gives detailed descriptions about the necessary transformations to perform BMC of software. The second part of the chapter presents some of the existing tools capable of performing BMC of software. Most of these tools were developed for C programs, however some of them are able to take as input an intermediate language representation.

Due to the success of CBMC, we give special attention to this tool. The existing documentation and papers about the tool, and more precisely about the transformations on the input program, do not show in detail how the generation of SA code is done. In order to understand how this generation works, we have made and documented an empirical study to CBMC. The

results of this study are presented in Chapter 4 and we believe this chapter is a useful contribution for readers wishing to understand how to produce SA code as a first step in the verification of software.

Chapter 5 is the core of this thesis. It is in this chapter that we present the main work of this thesis: *a bounded model checker for SPARK programs.* The first part of the chapter introduces the main technology being used in the implementation of our tool. One of the contributions described in this chapter is the development of a parser for SPARK programs. The current version of the parser supports the full SPARK language, and it is an open source project under the BSD 3 license, available from `https://bitbucket.org/vhaslab/spark-parser`. After that, we describe the internals of our tool. Taking the standard BMC transformations as guidance, we show how we have approached each step of the BMC workflow. We try to point out specific challenges of the SPARK language which required more work from our part. A tutorial about how to use the tool is also presented in that chapter, together with a set of examples.

The last chapter of this thesis is dedicated to presenting the conclusions and future work.

The work developed in this thesis has as context one of the tasks of the AVIACC[5] project, and has contributed directly to the project. Moreover, also in the context of the project, and using parts of this thesis, a communication [LMFP13] and a full paper [MFLP13] were presented at the 5th INForum ('Simpósio Nacional de Informática') which was held at Universidade de Évora.

[5] `http://wiki.di.uminho.pt/twiki/bin/view/Research/Aviacc/WebHome`

Chapter 2

SPARK

2.1 Language

Ada programming language can help avoid errors that are common in other languages but, as said before, in the development of highly critical systems this is often insufficient. SPARK [Bar03] appeared in 1988 to target precisely these systems. Although SPARK is based on a heavily restricted subset of Ada together with a set of annotations, it should be considered in its own right as a full language for the development of annotated high-assurance software. The great advantage of using a subset of a widely used language is that this makes possible to share compilers, instead of developing a new one. A consequence of this is that annotations in SPARK code must be written as Ada comments, ignored by compilers but not by the SPARK verification tools.

Figure 2.1 contains a procedure specification with the corresponding annotations (lines starting with `--#`). In both Ada and SPARK parameters can have one of three modes, indicating the direction of the information flow: `in`, `out` or `in out`. The example procedure has one parameter (`V`) with mode `in`, and one parameter (`M`) with mode `out`: it receives an array of integer values and returns the index of the maximum element. The specification includes a dataflow annotation (stating that the value of `M` derives from `V`) as well as a post-condition.

```
package Marray is
   Array_Size: constant:=10;
   subtype Index is Integer range 1 .. Array_Size;
   type VArray is array (Index) of Integer;

   procedure MaxArray(V: in VArray; M: out Index);
   --# derives M from V;
   --# post (for all I in Index => (V(I) <= V(M)));
end Marray;
```

```
package body Marray is
 procedure MaxArray(V: in VArray; M: out Index)
 is
   I: Integer;
   Max: Index;
 begin
   Max := Index'First;
   I := Index'First+1;
   loop
     exit when I > Index'Last;
     --# assert (for all J in Index range Index'First..(I-1)
     --#           => (V(J) <= V(Max)));
     --# assert (I >= Index'First) and (I <= Index'Last + 1);
     if V(I) > V(Max) then
        Max := I;
     end if;
     I := I + 1;
   end loop;
   M := Max;
 end MaxArray;
end MArray;
```

Figure 2.1: SPARK program: specification (top) and body (bottom)

A SPARK program is composed by one or more units. There exist two different kinds of program units: *packages* and *subprograms*. Subprograms define computations and are divided into *functions* and *procedures*. Functions are routines with only mode `in` parameters, that may not have side effects, and always return a value. Each function must have exactly one *return* statement, which must be the last statement (it cannot occur anywhere else in the function body). Function calls occur inside expressions, while procedure calls may only occur as standalone statements (procedure bodies may not contain return statements).

The other kind of subprogram unit, called package, is used as a way of grouping related entities (e.g. data types, data objects, subprograms or even nested packages).

All program units are generally divided in two parts: *specification* (the

program unit's interface) and *body* (the implementation details). Figure 2.1 contains both the package `Marray`'s specification (top) and body (bottom).

Many features of Ada are not present in SPARK because they are considered 'dangerous' in the development of *safety-critical systems*, or at least difficult to verify. These include recursion, dynamic memory allocation, access types (pointers), dynamic dispatching, and generics. See [Alt11] for a full description of the SPARK restrictions. Some of these exclusions facilitate our work in the development of a BMC for SPARK. For instance, unlike CBMC, a BMC for SPARK does not have to limit the number of times a subprogram is inlined, since recursion is not allowed. The same applies to pointer validity checks, since pointers are absent from SPARK.

SPARK is not just a language, but also a set of tools that not only check if a program respects all the restrictions imposed on valid SPARK programs, but are also probably the most widely used tools for program verification. The Examiner is the tool responsible for performing syntactic and static semantic analyses for checking the validity of SPARK programs, as well as generating verification conditions. In Section 2.2 we give a more complete description of the SPARK tools.

We now briefly describe the features to take into account in the development of a BMC for SPARK and to give an overview of the language.

Types

SPARK types are organized into categories [Bar03]. Figure 2.2 shows only the discrete types hierarchy (a subset of the types hierarchy), divided into *integer types* and *enumerations*. The integer types are divided into *signed integers* and *modular integers*. Operations over signed integer types may result in overflow (which raise runtime exceptions), whereas operations over modular types have wraparound semantics. A modular type is defined by giving a power of two integer `N`; its values range from zero to `N-1` (see type T in Figure 2.3). The range of the predefined integer type is defined in a default SPARK package, but it can also be given in a configuration file. It is also possible to define new integer types, with range given by two static

expressions (lower and upper bound). As an example, Figure 2.3 shows the declaration of the type `Nat`. Note the use of the expression `Integer'Last` which makes use of an attribute of the integer type. `S'Last` denotes in general the last element of `S`. Several attributes exists but in this section we only describe a subset of them. To have a full idea of every existing attribute refer to [Alt11].

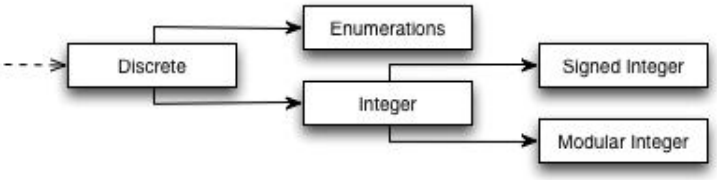


Figure 2.2: Discrete types

An enumeration is defined using a list of identifiers (enumeration literals). In SPARK, as opposed to Ada, enumeration literals cannot be overloaded. In general, elements of an enumeration are ordered as they were declared. Moreover, the equal, unequal and other relational operators may be used with enumerations. There are also some attributes which are very useful with enumerations. For instance, `Pos` (`Val`) returns the position of an enumeration element (the enumeration element corresponding to a position). `Pred` and `Succ` return the predecessor and successor of an element respectively. Type `Day` in Figure 2.3 is an example of a declared enumeration type; types `Boolean` and `Character` are predefined enumeration types. However, the `Boolean` type is a special case. Relational operations as well as the attributes `Pred`, `Succ`, `Pos`, `Val` do not apply to `Boolean` types. Equality and inequality as well as the attributes `First`, `Last` do apply to `Booleans`. As a consequence of these design choices, operations with boolean expressions never result in run-time exceptions.

In addition to discrete types there exist also *composite types*, divided into *records* and *arrays*. A record is a structure consisting of named components. As is the case in other languages, an array consists of an indexed list of elements of the same type. The index must be a discrete type (enumeration,

```
type Nat is range 0 .. Integer'Last;
type T is mod 4;
type Day is (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
subtype Counter is Nat range 1 .. 10;
subtype Weekday is Day range Day'First .. Fri;
```

```
type Tuple is array (Integer range <>) of Day;
type WorkHours is array (Weekday) of Nat;
type Matrix is array (Counter, Integer range <>) of Integer;
```

Figure 2.3: Examples of SPARK discrete and array types

integer or modular) and may possibly be *constrained*. In Figure 2.3, `Tuple` is an example of an unconstrained array declaration (note the use of `<>`) while `WorkHours` is an example of a constrained one. Objects of an array type must always have a static bound, therefore the types `Tuple` and `Matrix` cannot be used directly to create new objects, since they have unconstrained indexes.

A subtype of a certain type is defined by giving a lower bound and an upper bound over the base type. One can define subtypes of both enumeration types and integer types, but not of modular types or of the predefined `Boolean` type. In Figure 2.3, `Counter` and `Weekday` are examples of subtypes of the types `Nat` and `Day` respectively.

Statements

The SPARK language has the usual statements that are present in most programming languages. Assignments and if statements have the typical semantics as in other programming languages. The null statement is available through the keyword `null`. Case statements allow the use of simple values, alternatives and range expressions. Figure 2.4 shows an example of a case statement. The first option, `Mon .. Thurs`, represents a range: `Work` gets performed if `Today` evaluates to a value in this range. The second option tries to match `Today` with the value `Frid`. If it succeeds, both `Work` and `Party`, get performed. The last option represents an alternative, that is, if `Today` evaluates to `Sat` or `Sun`, the `null` statement is performed.

The most primitive form of iteration is implemented by an infinite `loop` control structure and an explicit abrupt `exit` command which should always

```
case Today is
  when Mon .. Thurs => Work;
  when Fri => Work; Party;
  when Sat | Sun => null;
end case;
```

Figure 2.4: Case statement in the SPARK language ([Bar03])

occur under an if statement without else branch, or alternatively within a `when` clause. In SPARK an exit statement always refers to the innermost loop. In Ada and SPARK is possible to name loop statements and to use these names in the exit statements (allows to exit nested loops in Ada), however, in SPARK, if the name is specified in the exit statement, such name must refer to the innermost loop. The loop in Figure 2.1 is an example of a loop with no iteration scheme. The language additionally provides two different iteration schemes: `while` loops and `for` loops. In SPARK, as opposed to Ada, it is not allowed to have a `for` loop without representing the type being iterated. For example the loop `for I in 1..10` is valid in Ada but not in SPARK. Instead, the type has to be specified, as for example `for I in S range 1..10`, for some integer type `S` containing the `range 1..10`.

Annotations

As was mentioned in the introduction, annotations in SPARK are grouped in two categories. While the core annotations are related to flow and visibility control, the proof annotations are concerned with formal proof.

The most important core annotations are the `global` and `derives` annotations. Both of them are used in procedures. `global` indicates that a global variable is used in the subprogram, and also specifies if the variable is read or written in the subprogram. Figure 2.5 shows an example taken from [Bar03] where a core annotation is used. It specifies that a global variable `Total` is read and written by the procedure `Add`. The `derives` annotation specifies the information flow between the input and output parameters, as well as global variables. In Figure 2.5 this is used to indicate that the final value of `Total` depends on the initial value of `Total` and `X`. Other core annotations

exist, related to visibility and control with respect to other packages and variables. We will not discuss these annotations, which are outside the scope of this thesis. Refer to [Bar03] for information about them.

```
procedure Add(X: in Integer);
--# global in out Total;
--# derives Total from Total, X;
```

Figure 2.5: Using core annotations ([Bar03])

The proof annotations are used to write functional specifications. They include pre-conditions and post-conditions, assertions such as loop invariants, and declarations of proof functions. These annotations are used to generate verification conditions to prove the safety and functional correctness of the program. Proof functions are not SPARK functions. Instead they are only declared in the proof context.

2.2 Toolset

Since there is no compiler for SPARK, there must exist a tool to check the validity of SPARK code, i.e. to verify if all restrictions imposed over a SPARK program are respected, and if the annotations, inserted as Ada comments, are valid. That tool is called *Examiner* and performs the checks mentioned before. Moreover, it uses the annotations to generate Verification Conditions (VCs) and to perform data and information flow analysis. This is the main tool of the SPARK language, and is the first one that must be used on any SPARK unit.

The VCs generated by the Examiner tool must be discharged, in the sense that they must be proved correct. For that, SPARK has both an automatic and an interactive proof tool. The automatic proof tool, called *Simplifier*, tries to automatically discharge VCs using predicate inference and rewriting. However, there are many VCs that Simplifier cannot discharge. For the latter, the interactive proof tool called *Proof Checker* may be used. The user works interactively with the tool in order to make progress proving some VC.

Such interaction requires already some level of expertise, which makes this task potentially very expensive.

Another important tool is the *Proof Obligation Summarizer (POGS)*, which creates a file with the summary containing the state of each VC. Figure 2.6 shows the interaction between the tools referred. As it is possible to see, Examiner takes as input the packages (specifications and bodies) and produces a file `vcg` which contains the VCs to be discharged. Those VCs are then passed to Simplifier, which tries to discharge them, and produces a file `siv`. This file is then used by the interactive proof tool (Proof Checker) for discharging other VCs, and to produce a file `plg`. Note also the *Review Team*, which consists of human reviewers that can also discharge VCs manually and write their status in the `prv` file. At the end, POGS takes the intermediate files and builds a summary with the state of each VC.

The tools mentioned before are those that have been used with SPARK for many years. However, lately other tools have been developed. *ZombieScope* is a tool capable of detecting dead paths, that is, paths that are never reached. It also works from files generated by the Examiner, and produces a file that is later read by POGS. Therefore, the summary of the ZombieScope result will also be available in the file generated by POGS.

There is also an available IDE for the development of SPARK applications. It is called GPS, and is provided by AdaCore. It has support for the language as well as the tools described above.

2.3 Examples

Let us first look with more attention at the example in Figure 2.1 focusing on some aspects not covered in the previous sections. The specification of `Marray` starts by declaring a numeric literal `Array_Size`, which has the value `10`. Note that the value of a numeric literal must be a static expression, that is, it must be evaluated in the same way independently of the context in which it is written (it cannot contain variables). The type `Index` and subtype `VArray` are just declared as explained in the previous section. Let us now look at the function annotations. The `derives` annotation is used

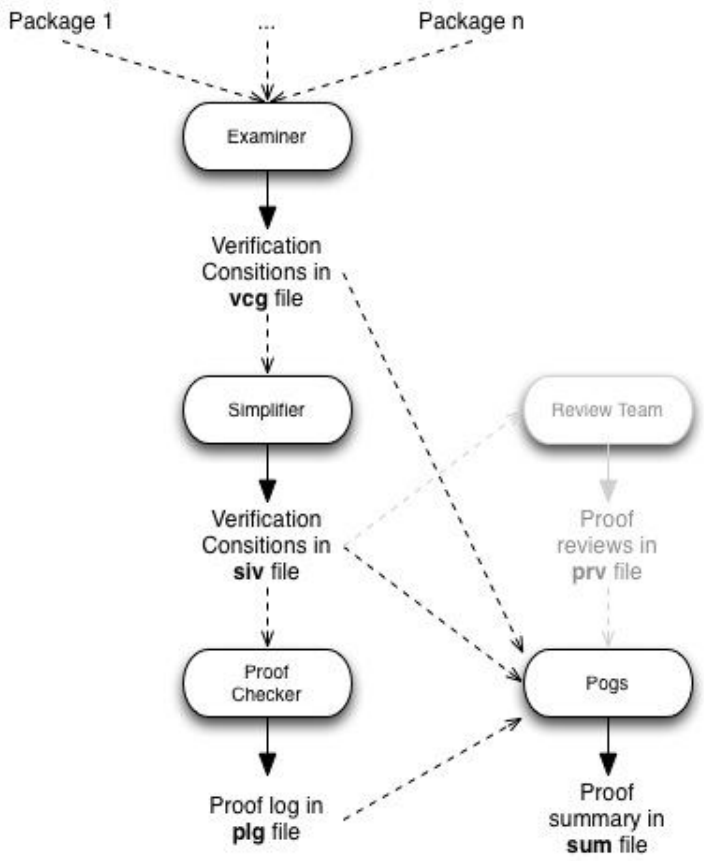


Figure 2.6: SPARK tools (inspired in [Bar03])

to indicate that the value of the output parameter `M` is calculated based on the value of `V`. The post-condition states that all elements are smaller than or equal to the element in the position returned in `M`, or in other words, it states that the index of the maximum value is returned.

For the VCs related to the post-condition be discharged, one has to write a loop invariant for the loop in the function body. The loop invariant says that all elements of the array whose index is smaller than the current index (`I`) must be smaller than or equal to the current maximum element. Moreover, for Simplifier to be able to prove the absence of under/overflow exceptions, another loop invariant is written stating that the value of `I` is always in the range between `Index'First` and `Index'Last + 1`. The Examiner will generate VCs to prove the correctness of the loop invariants, as well as of the

post-condition and safety properties such as overflow. These VCs may later be discharged by Simplifier of by the Proof Checker. The correctness of the `derives` annotation is verified by the Examiner tool.

```
package Factorial is
  --# function ffact(X: Integer) return Integer;

  function fact(X: Integer) return Integer;
  --# pre X >= 0;
  --# return R => R = ffact(X);
end Factorial;
```

```
package body Factorial is
  function fact(X: Integer) return Integer
  is
    F, I: Integer;
  begin
    F := 1;
    I := 1;
    loop
      --# assert I <= X + 1 and F = ffact(I-1);
      exit when ( I = X + 1 );
      F := F * I;
      I := I + 1;
    end loop;
    return F;
  end fact;
end Factorial;
```

Figure 2.7: Factorial in SPARK

As a second example, let us now analyze a program which calculates the factorial of a number given as input. Such an example is shown in Figure 2.7, and includes the package specification (top) and package body (bottom). In the specification we can see the pre-condition and post-condition as well as the declaration of a proof function [Bar03] `ffact`. Such a function is not a SPARK (or Ada) function, instead it is a function only used in the proof context. Theorems about `ffact` may have to be declared in order to allow VCs to be discharged. In this example there is also a pre-condition, which states that the input parameter has to be greater than or equal to zero (it does not make sense to calculate the factorial of a negative number). Moreover, for functions, instead of using a post-condition annotation, a `return` annotation must be used. In this case, it says that the return value must be equal to the result of applying the proof function to the input parameter.

Let us now focus on the body of the program, more precisely the annotation (everything else is just straightforward code, very similar to other languages). As in the previous example, in order to prove the return annotation (similar to the post-condition in the previous example) one has to write a loop invariant. Here, the invariant states that at every iteration `I` is always smaller than or equal to the input parameter plus one, and that `F` is the factorial of `I-1`. Note that, in order to discharge VCs related to overflow, the pre-condition has to be refined with the maximum number that can be calculated without causing overflow. This value depends on the range of the integer values (which must be specified as said before).

2.4 Industrial Use

The characteristics described above have helped make the SPARK language broadly used in critical software. Tokeneer is a nice example to explore, since it is publicly available[1]. With the intention of demonstrating that it was possible to develop software reaching Common Criteria EAL5[2], the National Security Agency asked Altran[3] to develop a part of Tokeneer. The project largely exceeded the EAL5, and the code was made available in 2009. Other uses of SPARK are described in [Cha00]. They include the Lockheed C130J (Hercules) plane, where SPARK was used in about 80% of the code for the Mission Computer. Two famous projects in which SPARK is being presently used are the CubeSat Lunar Lander/Orbiter Project[4] and the Alaska Ice Buoy Project[5]. Unfortunately, due to confidentially issues, detailed information about how exactly SPARK is being used in these projects (and many others) is not available.

[1] `http://www.adacore.com/sparkpro/tokeneer`
[2] `http://www.commoncriteriaportal.org/cc/`
[3] `http://www.altran.co.uk/`
[4] `http://www.cubesatlab.org/Lunari_Lander/Software_Components.shtml`
[5] `http://www.cubesatlab.org/Alaskan_Ice/Buoy_Software.shtml`

Chapter 3

Bounded Model Checking of Software: State of the Art

3.1 The BMC technique

The key idea of bounded model checking of software is to encode *bounded behaviors* of a program that enjoy some given property as a logical formula whose models, if any, describe executions leading to violations of the property [JM09]. The properties to be established are assertions on the program state, included in the program through the use of *assert statements*. For every execution of the program, whenever a statement `assert` ϕ is met, the assertion ϕ must be satisfied by the current state, otherwise it is said that the execution violates the assertion ϕ. This verification technique assumes that a satisfiability-checking tool is used to find models corresponding to property violations.

Observe that the interest of encoding bounded behaviors comes from the fact that the number of states grows exponentially with the number of variables and possibly with the length of the program, and it is in many cases infinite. Even if BMC cannot prove the (unbounded) correctness of a program, it is useful as long as it finds bugs that would otherwise be missed.

This section is dedicated to explaining the BMC workflow, which consists of automatically inserting safety properties in the source code program;

transforming the program in a way that it becomes easy to extract from it a logical formula that models bounded behaviors of the program; and finally checking for assertion violations by using a satisfiability solver.

The detailed description of the BMC steps are based on the study we have made of several publications and tools, [CKY03, IYG+08, XA05a, BH08, CFMS12]. The example in Figure 3.1 is used as a running example to demonstrate the transformations required to obtain such a model. The figure shows a fragment of C code, that calculates the index of the maximum value of an array. We assume that all variables are correctly declared and `max_array` contains the size of the array (greater than zero).

```
...
max = 0;
for(i = 1; i < max_array; i++){
  if (a[i] > a[max]) max = i;
}
...
```

Figure 3.1: Algorithm to calculate the index of the maximum element in an array

Inserting specific properties

Despite the idea of BMC of software being fully automatic, the user may provide a specification, using for that special annotations. These annotations are normally written as comments (not mandatory), so that they are ignored by a compiler. They provide the possibility for the user to write specific properties about the program that may not be automatically inserted. This may be very useful for debugging purposes or to check functional properties.

Two standard annotations are used in BMC of software: `assert` and `assume`. While the first imposes a restriction on the program, the second imposes a restriction on the model being built. An `assert p` states that `p` must be valid in a certain location of the program. When an `assert p` is annotated in the program, one wants the property `p` to be checked at that point of the program. If `p` does not hold, that violation is reported and a counter-example is given. On the other hand, an `assume p` states that one

can rely on the fact that `p` is true in a certain point of the program. Note that the property `p` is not checked but instead it is assumed as valid at that point of the program. If on a run of the program the property `p` is false, any property to be checked in the continuation of the program will be vacuously true, since in the logical model it will be implied by contradictory formulas. The `assume` annotation is particularly useful when applying a bound to the model. For instance it gives us the possibility to ignore all assertions outside the bounded model. Details about the encoding of these annotations will be given later.

A possible functional property over the program shown in Figure 3.1 would be for example to check if the maximum element is always in a user-predefined range. For that the user would have to write an `assert` exactly after the end of the `for` loop as follows:

```
...
max = 0;
for(i = 1; i < max_array; i++){
  if (a[i] > a[max]) max = i;
}
assert (min_range <= a[max] && a[max] <= max_range);
...
```

As said before, such annotations use to be written as comments so they can be ignored by compilers. However, in this section, for appearance purposes, we assume that the compiler knows these instructions and ignores them.

Similarly, the user can also ignore certain executions, using for that an `assume` annotation. In the example above, if `max_array` was an input parameter of a function, the user could for example state that its value is always greater than zero, writing the following after the function header:

```
...
assume (max_array > 0);
max = 0;
for(i = 1; i < max_array; i++){
  if (a[i] > a[max]) max = i;
}
...
```

Simplification

At the heart of a BMC tool stands an algorithm that extracts a logical formula directly from the source code (including properties expressed as assertions), without user intervention. The algorithm begins with some preliminary steps of transformation and simplification of the original program. These may include the removal of side effects (each statement is replaced by side-effect free statements with the same semantics), or normalization into a subset of the target programming language (say, a single loop form may be used if more than one is available). Reducing the number of different statements in an early stage, reduces the number of statements that have to be transformed at a later stage (which would probably require harder work). The previous example could be transformed as shown in Figure 3.2. Note that the `for` loop was written using a `while` loop and the side effect expression `i++;` was transformed into `i = i + 1;`. A more general approach to simplifying loops would be to use `goto` statements and `labels`, however, such an approach would make the code more difficult to read in the next steps, therefore in this section we will stick to while loops as the most general way of writing loops.

```
...
max = 0;
i = 1;
while(i < max_array){
  if (a[i] > a[max]) max = i;
  i = i + 1;
}
...
```

Figure 3.2: Simplification step

Automatic Instrumentation

Before a model is extracted from the source code, and in order to have a completely automatic verification tool, the code has to be annotated with safety properties that the user wants to check. This process, called instrumentation, can be fully automatic for most of the safety properties, as for

example overflow check, array out of bounds, null pointer access, division by zero, etc. This step can also occur before the simplification process, however, one would not take advantage of the simplification process.

Figure 3.3 shows our running example after the automatic instrumentation process. Assertions to check for overflow and array out of bounds access have been inserted. For the out of bounds assertions, it is enough to specify that the index expression is greater than or equal to zero and less than the size of the array. To check for overflow, a predicate `!overflow` is used. It receives the type resulting from an arithmetic operation, the operator and the operands. For more complicated expressions, one assertion would be inserted for each operator, to check if any internal expression would cause overflow. The use of a predicate to check for overflow is justified by the fact that for each type the minimum and maximum values are different, and depending on the background SAT or SMT solver the generated property may be written different.

```
...
max = 0;
i = 1;
while(i < max_array){
  assert (i >= 0) && (i < max_array);
  assert (max >= 0) && (max < max_array);
  if (a[i] > a[max]) max = i;
  assert !overflow(int, +, i, 1);
  i = i + 1;
}
...
```

Figure 3.3: Overflow and array out of bounds instrumentation

The bounded model

The next step is crucial – it is in this step that information is lost, in the sense that only bounded behaviors up to a limit are preserved. The remaining steps preserve all behaviors of the program. Given an entry-point provided by the user and a bound `K`, the program is expanded by unwinding loops a fixed number of times (loop bodies are replicated `K` times), and inlining routine calls (routine calls are replaced by routine bodies). Note that in the presence

of (mutual) recursion, a bound `K` is also imposed on the length of the inlining expansion. A program consisting of multiple routines is thus transformed into a monolithic program, which is both recursion-free and iteration-free. The idea is that executions of the simplified program correspond to finite prefixes of executions of the original program.

Once a loop or a recursive function call is unwound, one of two approaches may be followed, depending on whether the user wants to enforce soundness or just check a bounded model. To enforce soundness of bounded model checking, an `unwinding assertion` can be placed at the end of each expanded segment of code. This unwinding assertion is just an assert annotation stating that the loop was unwound a sufficient number of times to cover all the possible executions of the program. This way, if the unwinding assertion is not violated by any execution path, checking the transformed (bounded) code is sound, because the loop will never iterate more than `K` times.

When the interest of the user is to verify a bounded model (for instance, for bug finding) the approach to follow is to ignore assertions in executions beyond the bound. For that, an *unwinding assumption* may be inserted, instead of an unwinding assertion, to exclude properties which would require more iterations to be proved. In this case, one must always bear in mind the unsoundness of the approach.

The task of inserting an assertion to assure the loop was unwound a sufficient number of times, or an assumption to ignore properties requiring more than `K` iterations, is almost the same. First of all, to unwind a loop, the following rewriting rule must be applied `K` times:

$$\texttt{while(b) \{I\}} \longrightarrow \texttt{if(b) \{I; while(b) \{I\}\}}$$

and, after this, one of the following rewriting rules must be applied (depending on the desired behavior, following the previous discussion):

$\texttt{while(b) \{I\}} \longrightarrow \texttt{assert(!b);}$ $\qquad$ $\texttt{while(b) \{I\}} \longrightarrow \texttt{assume(!b);}$

The resulting code consists of replicated nested conditionals, where the inner conditional contains just an annotation (assert or assume) with the negation of the guard of the conditional.

`...if(b) {assert(!b);}...` `...if(b) {assume(!b);}...`

The `assert(!b)` or `assume(!b)` instruction is only reached if and only if `b` is evaluated to true, therefore, whenever the condition `!b` is evaluated, its result will be false, which allows us to write the second rewriting rule as follows:

`while(b) {I}` $\rightarrow$ `assert(false);` `while(b) {I}` $\rightarrow$ `assume(false);`

The idea is that whenever the assertion or the assumption is reached by an execution, the loop condition must be false. That is, the loop must have terminated before the assertion or the assumption was reached, otherwise such an execution requires more than `K` iterations. In the latter case, this execution is taken as a counter example (when an unwinding assertion is inserted) or else, the subsequent properties are trivially discharged (when an unwinding assumption is inserted).

Figure 3.4 shows the idea of unwinding a loop twice. On the left hand side, there is a loop with the condition `b`, and a loop body, that we identify by `loop_body`. Such loop body may have several instructions, including other loops that must also be recursively unwound. In the middle of the figure the loop was unwound twice and an assertion was used. On the right hand side of the figure, the loop was unwound twice, and an assume was used. As can be seen, the only difference at the syntax level is in the `assert` or `assume` keyword.

```
while(b){
  loop_body;
}
```

```
if(b){
  loop_body;
  if(b){
    loop_body;
    if(b){
      assert(false);
    }
  }
}
```

```
if(b){
  loop_body;
  if(b){
    loop_body;
    if(b){
      assume(false);
    }
  }
}
```

Figure 3.4: Loop unwinding 2x (unwinding assertion vs unwinding assumption)

Note that, either an unwinding assertion or an unwinding assumption should be inserted, otherwise, the approach of bounded model checking be-

comes both unsound and incomplete. Suppose that neither of them is inserted and there is an execution requiring more than K iterations. This execution would be considered for analysis, but the loop would iterate only K times and then exit abruptly, violating the loop exit condition. Now if this execution violates some property after the loop (inserted by the user or by an instrumentation tool), it will be considered as a counter example, but such counter example may not even exist in the original program. Figure 3.5 shows one example of the problem. Consider the loop in the left hand side of the figure, which gets expanded twice on the right hand side. The initial program uses the loop to decrement the variable i three times and then it adds three to the variable, therefore, no overflow ever occurs. However, on the right hand side of the figure, i gets decremented only two times. Thus, when the variable gets assigned with its value plus three, it will originate an overflow. If an unwinding assertion had been placed, the execution would be considered as counter example; and if an unwinding assumption had been placed, the property would be trivially discharged and no counter examples would be found.

```
i = MaxInt;
while(i > MaxInt - 3){
  i = i - 1;
}
assert !overflow(int, +, i, 3);
i = i + 3;
```

```
i = MaxInt;
if(i > MaxInt - 3){
  i = i - 1;
  if(i > MaxInt - 3){
    i = i - 1;
  }
}
assert !overflow(int, +, i, 3);
i = i + 3;
```

Figure 3.5: Making BMC of software not complete

Figure 3.6 shows the result of unwinding the loop in the program of Figure 3.3 twice, using the approach suggested above. Again the only difference from the left to the right is in the use of an unwinding assertion or an unwinding assumption.

```
...
max = 0;
i = 1;
if (i < max_array){
  assert (i >= 0) &&
    (i < max_array);
  assert (max >= 0) &&
    (max < max_array);
  if (a[i] > a[max]) max = i;
  assert !overflow(int, +, i, 1);
  i = i + 1;
  if (i < max_array){
    assert (i >= 0) &&
      (i < max_array);
    assert (max >= 0) &&
      (max < max_array);
    if (a[i] > a[max]) max = i;
    assert !overflow(int, +, i, 1);
    i = i + 1;
    if (i < max_array){
      assert false;
    }
  }
}
...
```

```
...
max = 0;
i = 1;
if (i < max_array){
  assert (i >= 0) &&
    (i < max_array);
  assert (max >= 0) &&
    (max < max_array);
  if (a[i] > a[max]) max = i;
  assert !overflow(int, +, i, 1);
  i = i + 1;
  if (i < max_array){
    assert (i >= 0) &&
      (i < max_array);
    assert (max >= 0) &&
      (max < max_array);
    if (a[i] > a[max]) max = i;
    assert !overflow(int, +, i, 1);
    i = i + 1;
    if (i < max_array){
      assume false;
    }
  }
}
...
```

Figure 3.6: Loop unwinding 2x, (left: unwinding assertion; right: unwinding assumption)

Single-assignment form

In order to extract a logical formula from an iteration-free and recursion-free program one has to first transform the program into a form in which the values of the variables do not change once they have been used (so that they can be seen as logical variables). This is done by converting the program into a Single Assignment (SA) form in which multiple indexed versions of each variable are used – a new version is introduced for each assignment to the original variable. A program is in the SA form if in every execution path, once a variable has been read or assigned it will not be assigned again.

SA has been around for many years, and it has been used mainly in the field of compilers design [Muc97] and more recently in verification techniques [BL05, dCFP12]. In its best-known form, called Static Single Assignment (SSA) [CFR+89], each variable may only appear once on the left hand side of an assignment. In the other form of SA, known as Dynamic Single Assignment (DSA), each variable may be assigned more than once, as long

as this happens in different paths. That is, for each possible execution of the program each variable only gets assigned once, but statically the variable may appear more than once on the left hand side of an assignment. We will tell more about transformation of a program to SA form in Chapter 4, where we describe in detail how C programs are annotated to SA form by the CBMC tool.

Having a program in SA form has some advantages in different applications. For instance many compiler optimization techniques are more efficient for programs in SA form. In the case of BMC this SA form of a program is closer to a logical model of the program. Since in SA form the value of a program variable is not changed, we can look at these variables as logical variables and assignments commands can be interpreted as logical equalities.

```
...
max1 = 0;
i1 = 1;
if (i1 < max_array1){
  assert (i1 >= 0) && (i1 < max_array);
  assert (max1 >= 0) && (max1 < max_array1);
  if (a1[i1] > a1[max1]) max2 = i1;
  max3 = (a1[i1] > a1[max1]) ? max2 : max2;
  assert !overflow(int, +, i1, 1);
  i2 = i1 + 1;
  if (i2 < max_array1){
    assert (i2 >= 0) && (i2 < max_array1);
    assert (max3 >= 0) && (max3 < max_array1);
    if (a1[i2] > a1[max3]) max4 = i2;
    max5 = (a1[i2] > a1[max3]) ? max4 : max3;
    assert !overflow(int, +, i2, 1);
    i3 = i2 + 1;
    if (i3 < max_array1){
      assert false;
    }
  }
  i4 = (i2 < max_array1) ? i3 : i2;
  max6 = (i2 < max_array1) ? max5 : max3;
}
i5 = (i1 < max_array1) ? i4 : i1;
max7 = (i1 < max_array1) ? i4 : i1;
...
```

Figure 3.7: Transformation into SA

Figure 3.7 shows our running example transformed into the SA form. Note in particular the use of the conditional expression, to assign the variables after a multi-branch statement (`if` statement in this case) to their ap-

propriate versions, thus synchronizing the variables used in both branches. The boolean expression used is the same used in the `if` statement. If the statement had an `else` clause the second value would be the last assigned in that clause, as in the following:

```
if b then
  x1 = ...
else
  x2 = ...
x3 = b ? x1 : x2;
```

Note also that the use of conditional expressions could be avoided by replacing each assignment with a conditional expression by an *if-then-else* in the obvious way.

Conditional Normal Form normalization

The next step in the BMC flow is to normalize the program into Conditional Normal Form (CNF). A program in CNF is a program consisting only in a sequence of single-branch conditional statements of the form `if` b `then` S, where S is an *atomic statement*, i.e. either an assignment, *assert*, or *assume* statement. Figure 3.8 shows the normalization of our running example. Note that nested `if` statements will be transformed in `if` structures in which the condition of the `if` is the conjunction of the conditions of the nested `if`s. The idea is that the branching structure of the program has now been flattened, so that every atomic statement is guarded by the conjunction of the conditions in the execution path leading to it.

Figure 3.8 shows our running example after the normalization into CNF. Note for example the assignment `max2 = i1` from the program shown in Figure 3.7, which is inside two nested `if` statements. After the normalization (Figure 3.8) this assignment is inside an `if` statement which contains the conjunction of both conditions.

Checking for property violations

At this point, two sets of logical formulas $\mathcal{C}$ and $\mathcal{P}$ can be extracted from the program. The set of formulas $\mathcal{C}$ describes logically the operational contents

```
...
if (True) max1 = 0;
if (True) i1 = 1;
if (i1 < max_array1) assert(i1 >= 0) && (i1 < max_array);
if (i1 < max_array1) assert(max1 >= 0) && (max1 < max_array1);
if (i1 < max_array1 && a1[i1] > a1[max1]) max2 = i1;
if (i1 < max_array1) max3 = (a1[i1] > a1[max1]) ? max2 : max2;
if (i1 < max_array1) assert !overflow(int, +, i1, 1);
if (i1 < max_array1) i2 = i1 + 1;
if (i1 < max_array1 && i2 < max_array1) assert(i2 >= 0) && (i2 < max_array1);
if (i1 < max_array1 && i2 < max_array1) assert(max3 >= 0)
                                        && (max3 < max_array1);
if (i1 < max_array1 && i2 < max_array1 && a1[i2] > a1[max3]) max4 = i2;
if (i1 < max_array1 && i2 < max_array1) max5 = (a1[i2] > a1[max3])?max4:max3;
if (i1 < max_array1 && i2 < max_array1) assert !overflow(int, +, i2, 1);
if (i1 < max_array1 && i2 < max_array1) i3 = i2 + 1;
if (i1 < max_array1 && i2 < max_array1 && i3 < max_array1) assert false;
if (i1 < max_array1) i4 = (i2 < max_array1) ? i3 : i2;
if (i1 < max_array1) max6 = (i2 < max_array1) ? max5 : max3;
if (True) i5 = (i1 < max_array1) ? i4 : i1;
if (True) max7 = (i1 < max_array1) ? i4 : i1;
...
```

Figure 3.8: Normalization into conditional normal form

of the program, and P contains the properties to be established. While $\mathcal{C}$ includes a formula $b \rightarrow x = e$ for every statement `if` b `then` $x := e$ contained in the CNF, the set $\mathcal{P}$ is extracted from the guarded assert and assume statements of the CNF. Recall that `assert p` states that the property `p` must be checked at that point of the program and `assume p` states that `p` is assumed as being true at that point of the program. While the asserted properties are the ones to be checked, the assumed properties are not to be checked, but can be useful to guarantee properties asserted subsequently in the program. The logical encoding of each (`if` b `then assume` θ) statement of the CNF program is the formula $b \rightarrow \theta$. In logical terms, each (`if` b `then assert` ϕ) statement of the CNF program is represented by the formula $(\bigwedge \mathcal{A}) \wedge b \rightarrow \phi$, where $\mathcal{A}$ is the set of all the $b \rightarrow \theta$ formulas assumed before.

Figure 3.9 shows an example where assert and assume annotations are used. This program would originate the following set of formulas $\mathcal{C}$ and $\mathcal{P}$:

$$\begin{aligned} \mathcal{C} &= \{x1 = y1, z1 = 10, b \rightarrow x2 = x1 + y1, \neg b \rightarrow z2 = x1, \\ &\qquad x3 = b?x2 : x1, z3 = b?z2 : z1\} \\ \mathcal{P} &= \{\phi_1, \theta_1 \rightarrow \phi_2, \theta_1 \wedge b \rightarrow \phi_3, \theta_1 \wedge \neg b \rightarrow \phi_4, \theta_1 \rightarrow \phi_5\} \end{aligned}$$

```
assert φ1;
x1 = y1;
assume θ1;
assert φ2;
z1 = 10;
if b then{
  x2 = x1 + y1;
  assert φ3;
} else {
  z2 = x1;
  assert φ4;
}
x3 = b ? x2 : x1;
z3 = b ? z2 : z1;
assert φ5;
```

Figure 3.9: Simple example with assert and assume annotations

From here it is simple to see that an `assume p` where `p` evaluates to `False` will cause all subsequent assert statements to be trivially discharged, since $False \rightarrow b = True$.

For our running example (shown in Figure 3.8) the task of extracting the set of formulas $\mathcal{C}$ and $\mathcal{P}$ is trivial, since there is not any assume annotation. Every statement containing an assert goes to $\mathcal{P}$ the others go to $\mathcal{C}$.

If no assertion from $\mathcal{P}$ fails in any execution of the program one has that $\bigwedge \mathcal{P}$ is a logical consequence of $\mathcal{C}$. This can be determined by checking the satisfiability of the set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$. Any model found for it corresponds to an execution that leads to an assertion violation. Of course, satisfiability checking is restricted to models that capture the properties of the data structures manipulated by the program, and that are specified by some background theory $\mathcal{T}$ (usually a combination of several logical theories). Therefore 'satisfiability' should in fact be understood as $\mathcal{T}$-satisfiability.

Satisfiability checks can be made either by a SAT solver [GKSS08] or an SMT solver [dMB11]: the main difference is the way in which numeric values and arrays are modelled. When using a SAT solver, numeric values are necessarily modelled as vectors of bits of fixed size, and each array element as a different variable. When using an SMT solver, numeric values can be modelled either as bit-vectors or as values in the semantic theory of the appropriate type (e.g. unbounded integers or reals), and arrays are modelled

as arrays (i.e. by a theory of arrays). This makes the SMT encoding independent of the size of arrays occurring in the program, which is not the case when using a SAT encoding. The use of bit-vectors captures precisely the low-level fixed-width machine semantics of program data types, and is very appropriate for capturing the 'silent overflow' behavior of programs. This will be particularly significant in the treatment of SPARK data types, as we will see in Chapter 5.

3.2 Existing Tools Based on BMC

There are several tools performing BMC of software. We use this section to give an overview of them. CBMC was the first to appear and is, very likely, the most successful one. Other tools were presented over the years using different approaches, whether at the transformation phase or at the encoding of logical formulas phase. Note that CBMC is the great motivation for this work, and therefore we devote most of the section to it. The other tools are presented for comparison purposes and to observe what they provide that CBMC does not.

3.2.1 CBMC

The CBMC tool was created in 2003 at CMU with the aim of checking the consistency between an ANSI-C program and a circuit given in Verilog [CKY03]. This verification was done using bounded model checking, unwinding the ANSI-C program and the circuit implementation and then translating it to a boolean formula. In 2004 CBMC appears as a tool for checking ANSI-C programs [CKL04] with support for the initial aim (consistency between ANSI-C and Verilog). Nowadays, CBMC is able to perform bounded model checking both of C and C++ programs.

The tool allows several properties to be checked in a completely automatic form. It can emulate different architectural environments for the program being analyzed. Little-endian and big-endian memory organization are both supported, and header files for linux, windows and mac os x are also available.

The width of the `int` type may also be specified manually.

CBMC is divided into three separated tools: *goto-cc*, *goto-instrument*, and *cbmc*; Note that if cbmc is invoked with an ANSI-C program, those tools will run in the background, transparently to the user. Next we give a brief description of each tool.

Pre-compilation: goto-cc

The goto-cc tool acts as a pre-compiler that takes a (.c) file (maybe containing user annotations) as input and generates a goto binary program. It is responsible for expanding all the *#define* directives, replacing side effects expressions by equivalent assignments (see [CKY03]), and transform *for/while* loops, *breaks* and *continues* statements into goto statements. Figure 3.10 (right) shows the transformations after goto-cc is performed on the program shown in Figure 3.10 (left). Basically this tool corresponds to the simplification step as described in Section 3.1.

```
#define A 3;

int main(){
  int i, j = A;
  for(i=0;i<3;i++){
    j = j + i;
  }
  return 0;
}
```

```
main (c::main):
    int i;
    int j;
    j = 3;
    i = 0;
 1: IF !(i < 3) THEN GOTO 2
    j = j + i;
    i = i + 1;
    GOTO 1
 2: return 0;
```

Figure 3.10: Applying goto-cc to an ANSI-C program

Instrumentation: goto-instrument

The goto-instrument tool corresponds to the instrumentation step as described in Section 3.1. It takes the file generated previously and automatically adds general assertions to prove certain safety properties. The properties available are: array out of bounds access, division by zero, null pointers dereference, signed and unsigned overflow/underflow check, uninitialized variables, unreachable labels and 'not a number' (NaN) occurrences. When

using CBMC (or goto-instrument), the user must specify which properties must be verified. For example, if the user chooses to check for overflow, an assertion is inserted before each arithmetic operation to check if it originates overflow. Figure 3.11 shows the result of using goto-instrument on the previous example program to check for overflow.

```
main (c::main):
     int i;
     int j;
     j = 3;
     i = 0;
  1: IF !(i < 3) THEN GOTO 2
     assert !overflow("+", int, j, i) // arithmetic overflow on +
     j = j + i;
     assert !overflow("+", int, i, 1) // arithmetic overflow on +
     i = i + 1;
     GOTO 1
  2: return 0;
```

Figure 3.11: Result of running goto-instrument with overflow check

Transformations and verification: cbmc

The tool cbmc is responsible for all other BMC transformations after the instrumentation. Using the file generated by goto-instrument all loops (which are in the form of goto) are unwound, replicating the loop body `K` times. This value, `K`, may be given by the user when invoking the tool, or, if it is not, cbmc tries to unwind the loop a necessary number of times, in order to obtain a sound approach. If cbmc is not able to infer `K` the loop is unwound infinitely while there are resources available. Each copy of the loop body is guarded by an *if* statement using the same condition as the loop. This happens for the case in which the loop requires less iterations than the times the loop has been unwound. After the last replicated body, an unwinding assertion or an assumption is placed, as explained in the previous section. Function calls are also inlined and in the case of a recursive function call, a bound is also applied as in the loop statements. Applying these transformations to the previous example would produce the program shown in Figure 3.12.

The result of the previous operations consists only of *if* statements, annotations, and assignments. The next step is to transform the program into

```
j = 3
i = 0
assert(i < 3 => !overflow("+", signed int, j, i))
IF (i < 3) j = j + i
assert(i < 3 => !overflow("+", signed int, i, 1))
IF (i < 3) i = i + 1
assert(i < 3 => !overflow("+", signed int, j, i))
IF (i < 3) j = j + i
assert(i < 3 => !overflow("+", signed int, i, 1))
IF (i < 3) i = i + 1
assert(i < 3 => !overflow("+", signed int, j, i))
IF (i < 3) j = j + i
assert(i < 3 => !overflow("+", signed int, i, 1))
IF (i < 3) i = i + 1
assert(!(i < 3))
```

Figure 3.12: Loop unwind with unwinding assertion in CBMC

an equivalent SA program, which will be explained in detail in Chapter 4. After the *SA* form is computed, two bit-vector equations are created and sent to a solver as explained in Section 3.1.

Since its introduction CBMC relies on a SAT solver, more precisely on MiniSat [ES03], for looking for situations where assertions stated in the program do not hold. Nevertheless, an encoding in the DIMACS format (recognized by most SAT solvers) can also be generated by the tool and checked for satisfiability using other solvers. The tool also has support for SMT solvers, however this line of work is still experimental. By default, Z3 [dMB08] is used when the SMT flag is used. Support for Boolector [BB09], MathSAT [BCF+08] and CVC3 [BT07] is also available, and a logical encoding in the SMT-LIB[1] version 1 or 2 can also be obtained.

CBMC is being applied in several fields. Official web page[2] has a list with its application, which includes: error explanation and localization, concurrency, cyper-physical systems, test-vector generation, etc.

[1] `http://www.smt-lib.org/`

[2] `http://www.cprover.org/cbmc/applications.shtml`

3.2.2 F-SOFT

F-SOFT was developed at NEC Laboratories[3] and was first presented in 2004 [IYG+04]. It is important to refer that the tool is only used internally at the NEC Laboratories and therefore is not available for public usage. All the information presented here is taken from published work. The biggest difference of this tool when compared to CBMC is the possibility of performing also unbounded model checking of software. However, due to the theme of this thesis we will focus exclusively on the BMC part of the tool. Information about the whole picture can be found in [IYG+08].

The tool starts by doing some transformations on the initial program using a C Intermediate Language (CIL) tool [NMRW02], which converts side-effect expressions into side-effect-free expressions, renames variables to be globally unique, and transforms complex C constructs into simpler ones, for example *while loops*, *break* and *continue* statements are all transformed into *goto* statements. F-SOFT does not unwind loops. Instead, it builds a Labeled Transition Graph (LTG) and when performing BMC it only considers paths whose length is not bigger than an user-provided bound `K`. Since the resulting boolean model consists of a symbolic transition system, F-SOFT does not require the program to be transformed into a SA form. At each state a configuration of the system at a certain location is kept. The program flow is described through transitions. After this first transformation, program slicing and range analysis are performed. Program slicing consists in removing all the elements that do not affect the property being verified. Range analysis consists in limiting the number of bits used to represent each variable in the boolean model. These transformations are common to *model checking* and BMC.

For the verification, the back-end tool VeriSol (based on DiVer [GGA05]) is used. If VeriSol returns a counter-example, an executable program is built and returned to the user, so it can be used with a debugger. According to the authors, a tool based on F-SOFT called VARVEL [IBG+11] is being used

[3] `http://www.nec-labs.com/`

at NEC [4], and it also supports C++.

3.2.3 Saturn

The first paper about Saturn was presented in early 2005 [XA05b] and it was part of a PhD thesis [Xie06] (Standford University). The main goal of SATURN is scalability, in particular it has been applied to the Linux kernel (more than 6MLOC) and other open source projects. It is important to refer that the tool is presented as a bug finder and not as a verification framework [XA07]. As opposed to the other tools where it is possible to conclude that a program is correct relatively to a property up to a bound, this does not happen in Saturn. Also the counter-examples returned by the tool are not necessarily valid, that is, the tool is neither complete nor sound. This is due to the design choices to reach scalability, as is explained next.

As in F-SOFT, the first step is to convert the C source program into a CIL representation, and from this representation extract the Abstract Syntax Tree (AST), which is then used to build function summaries. Function summaries are the main characteristic of this tool. Saturn verifies one function at a time, replacing function calls by function summaries. This summary can be seen somewhat as an abstraction of the model. Therefore, whenever a counter-example is found, it may well not exist in the real program. This is the reason why the tool is not complete. On the other hand, with this idea of summaries, it is possible to parallelize the verification of several functions at the same time. Computer clusters with 40 to 100 cores have been used for the verification, and the efficiency achieved was around 80-90% [ABDD07] when compared to single core machines. Loops are unwound a small number of times (twice in the Linux kernel lock checker case study, [ABDD07]).

Saturn does not provide an automatic instrumentation tool. Nevertheless, annotations as those presented in the previous section may be used to write properties to be verified.

[4] `http://www.nec.com/`

3.2.4 CALYSTO

CALYSTO was first presented in 2007 [BH07] and has been part of a PhD thesis [Bab08] (University Of British Columbia). According to the authors, the tool is based on ESC/JAVA [FLL+02], CBMC and especially on Saturn. Following [BH08], the tool scales better than CBMC and it is more precise than Saturn (relatively to the spurious counter-examples and number of bugs found), keeping the automation of both tools.

The tool works on the LLVM IR* representation [LA04], which transforms the input code into SA form and has the advantage of supporting many programming languages, requiring for that only a different front-end. The implementation is very similar to Saturn. The strong point about CALYSTO is that when a counter-example is found, if it depends on a specific function summary, this function summary is replaced by the inlined function. So, the functions are incrementally inlined during the analysis. The tool also has its own prover, SPEAR, which makes this process possible. Moreover, the prover is optimized to verify the Verification Conditions (VCs) generated by CALYSTO.

CALYSTO is able to verify any user-defined assertions, and it is able to automatically generate VCs for null pointer dereferences. Although the initial idea was to create an instrumentation tool as that presented in CBMC, according to the information available from the tool web page[5] the tool is not being updated for a long time.

3.2.5 ESBMC

ESBMC was first proposed in 2009 during the International Conference on Automated Software Engineering. It is a joint project between University of Southampton, University of Stellenbosch and Federal University of Amazonas. This tool performs BMC of software and it is completely devoted to taking advantage of SMT theories.

The tool is developed on top of CBMC. Therefore, it takes all the machinery to parse and simplify a program, as well as, to instrument and transform

[5] `http://www.domagoj-babic.com/index.php/ResearchProjects/Calysto`

it until the logical formulas that models the program are reached. However, at this point these sets of formulas are encoded using the new approach of ESBMC. This encoding makes use of optimizations like for example constant propagation and forward substitution techniques [Muc97]. The tool allows the user to choose between unbounded integers or bit-vectors for the representation of integer values. Enumerations are encoded as if they were integer values. For floating points, only fixed-point arithmetic is supported. The tool is also able to encode pointers and dynamic memory allocation. The SMT solvers supported by ESBMC at the time of writing this dissertation are: Boolector [BB09], CVC3 [BT07], and Z3 [dMB08]. Moreover, encodings using SMT-LIB version 1 or 2 may also be generated.

3.2.6 LLBMC

LLBMC was developed at Karlsruhe Institute of Technology (KIT), and was first presented in 2012 at the 4th International Conference on Verified Software [MFS12]. The tool was aimed at verifying C/C++ programs, however, instead of manipulating C/C++ source code, it uses LLVM to compile the program into LLVM intermediate representation, and then it applies the BMC transformations to this representation. Using such an approach has many advantages: the LLVM intermediate representation has a much clearer syntax and semantics than C/C++ code; the intermediate representation program is much closer to the program that is actually executed (because of optimizations, for example); it is possible to use this BMC with other languages provided that LLVM supports them.

The tool is able to instrument the code for integer overflow, division by zero and invalid shifting. Moreover, the tool is also able to deal with pointers and dynamic memory allocation, more precisely the tool is able to insert and check annotations related to illegal memory access (array index out of bounds or illegal pointer access) and invalid uses of `free`. As back-end solver, LLBMC uses Boolector [BB09] as default, but it also supports Z3 [dMB08] and STP [GD07].

The problem with this tool is the lifting of the error from the intermediate

representation back to C/C++ code. The authors claim that work has been done in this direction, but at the time of writing this dissertation there were no announcements of any progress.

Chapter 4

SA Generation in CBMC

As was seen before, one of the steps in the implementation of BMC of software consists of tranlating the program into an equivalent program in SA form. On the other hand, CBMC was one the first tools to be proposed performing BMC of software, and it is nowadays well known for its success. In this chapter, we explore how CBMC transforms a program into SA form.

We make two important remarks: although it is more or less clear what the SA form of a simple block of code is, this is less so far for full C programs consisting of different files and multiple functions, with every kind of variable. In addition, the internal SA representation used by a verification tool like CBMC is not the same as a standard SA form used in compilation workflow. So it is essential to understand exactly the different aspects introduced by CBMC in the SA transformation code, if we are to develop our own BMC tool for a different programming language.

Since the tool does not have any default way of showing the internal code in the SA form, we have reverse engenieered the process by:

1. writing simple C programs using different functionalities and variables contexts;

2. using a functionality of CBMC, to generate a file in the SMT-LIB version 2 format;

3. using the initial program and the generated SMT-LIB file to build the

program in the SA form.

Next we show some of the examples used to follow this approach. Each example aims to answer a specific aspect of the language, as for example different variable contexts, function inlining, etc. We do not give details about how the SA program is built, but instead focus on the analysis of the SA program to understand the way CBMC renames variables. To this end we show the initial program as well the program written in the SA form, explaining the tactic used to rename each variable. For each example, we ran CBMC as follow:

```
cbmc --signed-overflow-check --smt2 --outfile testX_smt2
     --no-propagation testX.c
```

Note that we have used CBMC version 4.1. The flag `--signed-overflow-check` is used to instruct CBMC to introduce assertions to check for overflow, otherwise, with the slicing functionality of CBMC, the resulting formula would be empty because there would be no assertions to prove (the slicing functionality removes every statement that does not influence the assertions to prove). Flags `--smt2 --outfile testX_smt2` are used to generate a file called `testX_smt2`, containing the model in the SMT-LIB version 2 format; the flag `--no-propagation` is used to avoid the propagation of variables; `testX.c` is the input file name.

4.1 Global Variables

The purpose of this first example is to explore how CBMC renames global variables. In the program shown in Figure 4.1 (left) there are only four global variables and a function with some arithmetic expressions manipulating these variables. The program has nothing complicated, but in the SA form obtained (shown on the right hand side of the figure) it is already possible to observe how CBMC deals with global variables. For each global variable occurrence, an index is appended to the name counting the number of times it has been assigned so far. Note also the particular default behavior

of CBMC, assigning the value `0` to each non initialized variable, as described in [KC06].

```
int x,y,w,z;

void test1 (void) {
  x = x + w;
  y = x + 10;
  z = 10;
  z = z + 10;
}
```

```
x#1 = 0;
y#1 = 0;
w#1 = 0;
z#1 = 0;
x#2 = w#1 + x#1;
y#2 = x#2 + 10;
z#2 = 10
z#3 = z#2 + 10;
```

Figure 4.1: Renaming global variables

4.2 Local Variables

Once we understand how global variables are handled, we mix global and local variables, as well as function parameters, in a single program. Figure 4.2 shows a program with a global variable `z`, two function parameters `x` and `w`, and one local variable `y`.

```
int z;

void test2(int x, int w){
  int y;
  x = x + w + y;
  y = x + 10;
  z = 10;
  z = z + 10;
}
```

```
z#1 = 0
test2::x!0@1#1 =
  nondet_symex::nondet0
test2::w!0@1#1 =
  nondet_symex::nondet1
test2::x!0@1#2 =
  test2::x!0@1#1 +
  test2::w!0@1#1 +
  test2::1::y!0@1#1
test2::1::y!0@1#2 =
   test2::x!0@1#2 + 10
z#2 = 10
z#3 = z#2 + 10
```

Figure 4.2: Renaming function parameters, local and global variables

As it is possible to observe on the right hand side of the figure, global variables are handled exactly the same way as in the previous example. With respect to function arguments, first of all observe that they are initialized with `nondet_symex::nondet`. This can be seen as a non-deterministic function, which returns a random value. Both local variables and parameters are

renamed resorting to the container function name (`test2`) and some counters. Each counter is preceded by a symbol. The counter next to `@` counts the number of times the function has been called so far, if we think in terms of the original program, or it can also be seen as the number of times it has been inlined, if we think in terms of the transformation (to reach such conclusion, other examples have been used). The counter after `#` counts the number of times the variable has been assigned so far in the same function call. The value next to `!` was, and remains a mystery during our experiments with the tool. It has been `0` with all the examples we could think of.

Note also the value `1` surrounded by the symbol `::` in the local variables. For now let us assume it as an identifier for local variables. Later its real meaning will be shown. Note also that non initialized local variables are not initialized with 0 as opposed to global variables.

Focusing on the assignment `test2::1::y!0@1#2 = test2::x!0@1#2+ 10`, which corresponds to the assignment `y = x + 10`, we can see by the counter after @ that both variables belong to the first function call, i.e. the first time the function `test2` was inlined. Moreover, the value after `#` indicates the version of the variable inside the same function call, that is, the number of times it has been assigned so far. Therefore, it is known that version 2 of the variable `x` is being used, which is assigned immediately before. Moreover, if we look blindly to the assignment we could imagine that variable `y` is being assigned for the second time, which **is actually false!** This is the first time the variable is assigned, however, maybe related to implementation issues, this counter on local variables starts at `2`. This brings no problem at all, as soon as it introduces a new version of the variable.

4.3 Function Calls

With the example in Figure 4.3 we analyze how CBMC renames variables when there are more than one function and there exist function calls and return statements inside functions. In the example there are four global variables, `x`, `y`, `w` and `z` and two functions, `test2` and `test4` with the respective local variables `y` and `x`, `v`. Moreover, function `test2` has two parameters `x`

and `w`.

```
int x,y,w,z;

int test2(int x, int w) {
  int y;
  x = x + w + y;
  y = x + 10;
  z = 10;
  z = z + 10;
  return y;
}

void test4 (void) {
  int x = 15;
  x = test2(x, 10);
  int v = x + z;
}
```

```
x#1 = 0
y#1 = 0
w#1 = 0
z#1 = 0
test4::1::x!0@1#2 = 15
test2::x!0@1#1 =
  test4::1::x!0@1#2
test2::w!0@1#1 = 10
test2::x!0@1#2 =
  test2::x!0@1#1 +
  test2::w!0@1#1 +
  test2::1::y!0@1#1
test2::1::y!0@1#2 =
  test2::x!0@1#2 + 10
z#2 = 10
z#3 = z#2 + 10
test4::1::x!0@1#3 =
  test2::1::y!0@1#2
test4::1::v!0@1#2 =
  test4::1::x!0@1#3 +
  z#3
```

Figure 4.3: Function calls with parameters

The entry point for our example will be `test4`, which has a function call to the function `test2`. For this example a special switch (`--function test4`) indicating the entry point (`test4` in this case) was used when invoking CBMC. CBMC inlines the code corresponding to the function call `test2(x,10)` in the assignment `x = test2(x,10)`, which places this code immediately before the assignment and propagates the return value with an assignment between variables (we have analyzed that sometimes CBMC uses a temporal variable to propagate the return value). The parameter passing is handled by using assignments directly to the variables. With respect to the renaming of variables, this example shows us what has been explained before, but now involving more than one function. Global variables are initialized at zero and only one counter is added to the name. For local variables two important counters are used: one counting the number of calls, the other counting the number of assignments. For instance the instruction `x = test2(x,10)` in `test4` becomes `test4::1::x!0@1#3 = test2::1::y!0@1#2`, where variable `x` is assigned with the return value of function `test2`.

4.4 Scopes

So long we have seen the behavior of global variables and local variables (including function parameters). Global variables are visible in the whole program, while local variables are only visible inside the function where they are declared. The generalization is that a C program is composed by scopes. Inner scopes inherit variables from outer scopes. Inside an inner scope it is possible to declare variables with the same name as variables occurring in the outer scope, which will hide the outer scope variables.

```
int z,k=100;

void f(int x, int w) {
  int y ;
  x = x + w + y;
  {
    int y = 10;
    int x = 20;
    int z = 50;
    x = x + y + z + k;
    {
      int y = 12345;
      x = x + y;
    }
    {
      int z = 9876;
    }
  }
  y = x + 10;
}
```

```
k#1 = 100
z#1 = 0
f::x!0@1#1 =
  nondet_symex::nondet0
f::w!0@1#1 =
  nondet_symex::nondet1
f::x!0@1#2 =
  f::x!0@1#1 +
  f::w!0@1#1 +
f::1::y!0@1#1
f::1::1::y!0@1#2 = 10
f::1::1::x!0@1#2 = 20
f::1::1::z!0@1#2 = 50
f::1::1::x!0@1#3 =
  f::1::1::x!0@1#2 +
  f::1::1::y!0@1#2 +
  f::1::1::z!0@1#2 + k#1
f::1::1::1::y!0@1#2 = 12345
f::1::1::x!0@1#4 =
  f::1::1::x!0@1#3 +
  f::1::1::1::y!0@1#2
f::1::1::2::z!0@1#2 = 9876;
f::1::y!0@1#2 = f::x!0@1#2 + 10
```

Figure 4.4: Renaming variables in different scopes

The idea of the example of Figure 4.4 is to show how CBMC deals with scopes. In this example there are two global variables `z` and `k`, a function called `f` which has two parameters `x` and `w` and a local variable `y`, and there are also different scopes inside the function, which declare new variables.

With this example it is possible to see that what was until now seemed to be the distinction between function parameters and local variables (`::1`) is actually a mechanism identifying the scope. Actually, the number after `::` is also a counter. It counts the number of declared scopes at the same level. At the function level there is one scope in which every declared variable has the

prefix `f::1::1`. The first part indicates that the variable is somewhere inside the function. The second part indicates that it was declared inside a scope declared at the function level. Moreover, inside this scope, two new scopes are declared. The variables in the first scope have the prefix `f::1::1::1`, while the variables declared inside the second scope have the prefix `f::1::1::2`.

To sum up, with this example it is possible to understand the way in which CBMC deals with scopes: it adds a substring at each level, which in turn, contains a counter for the case in which multiple scopes are declared.

4.5 Static Variables

In the C language, besides the global variables and local variables, which include variables declared in inner scopes, there exist also static variables that have a different visibility scope. The scope of a static variable is the file in which it is declared. In the example of Figure 4.5 we explore how CBMC handles static variables. In this example there are several files: *testeStatic.c*, *aux.h*, *aux.c*, *aux1.h*, and *aux1.c*. The header files are not shown here, since they just declare the function defined in the '.c' file. All files declare a static variable sV.

On the right side we can see that static variables, in the SA form, are treated almost the same way as global variables. The difference is that a numbering is used to differentiate them in the different files. The order in which they are enumerated is unknown but also not relevant provided that they are different. In this example, the first occurrence, `sV#1`, belongs to *aux.c*, the second, `sV#link0#1` belongs to *aux1.c*, and the last, `sV#link1#1` belongs to *testeStatic.c*.

4.6 Arrays

Let us now explore how arrays are handled by CBMC. The example in Figure 4.6 shows a single function, `f`, which starts by declaring an array `a`, and an uninitialized variable `i`. Some operations are performed involving the ar-

```c
//file aux.c
static int sV = 123;
int f(int a){
  int i = 987;
  i = sV + 876;
  return i + a + sV;
}
```

```c
//file aux1.c
static int sV = 234;

int g(int b){
  int i = 987;
  i = sV + 876;
  return i + b + sV;
}
```

```c
//file testeStatic.c
#include"aux.h"
#include"aux1.h"

static int sV = 0;
int i;

int main(){
  i = 10;
  sV = i;
  sV = sV + 200;
  int j = i + sV;
  j = j + f(i) + g(j);
}
```

```
sV#1 = 123
sV#link0#1 = 234
i#1 = 0
sV#link1#1 = 0
i#2 = 10
sV#link1#2 = i#2
sV#link1#3 = sV#link1#2 + 200
main::1::j!0@2#1 = i#2 + sV#link1#3
f::a!0@1#1 = i#2
f::1::i!0@2#1 = 987
f::1::i!0@2#2 = sV#1 + 876
main::$tmp::return_value_f$1!0@2#1 =
  f::1::i!0@2#2 + f::a!0@1#1 + sV#1
g::b!0@1#1 = main::1::j!0@2#1
g::1::i!0@2#1 = 987
g::1::i!0@2#2 = sV#link0#1 + 876
main::$tmp::return_value_g$2!0@2#1 =
  g::1::i!0@2#2 + g::b!0@1#1 +
  sV#link0#1
main::1::j!0@2#2 =
  main::1::j!0@2#1 +
  main::$tmp::return_value_f$1!0@2#1 +
  main::$tmp::return_value_g$2!0@2#1
```

Figure 4.5: Renaming static variables

ray, and then the sum of the first element with the second is returned. Note that the loop was unwound three times (`K = 3`), which is enough to obtain a sound model.

Let us observe the obtained SA form: the first assignment derives from the `for` loop initialization. The second assignment that derives from the assignment `a[i] = i` brings something new that must be taken into consideration. Since the goal is to obtain a logical encoding, arrays are modeled as applicative data structures with operators `select` (that returns the value stored in a given position of the array) and `store` (that stores a value in a given position of the array). That is, an assignment to an element of an array is transformed into an assignment to a simple variable of a value constructed from the previous value of that variable, using a dedicated function. That function, called

```
int f(){
  int a[3];
  int i;
  for(i=0;i<3;i++){
    a[i] = i;
  }
  int b = a[0] + a[1];
  return b;
}
```

```
f::1::i!0@1#2 = 0
f::1::a!0@1#2 =
  store(f::1::a!0@1#1,
        f::1::i!0@1#2,
        f::1::i!0@1#2)
f::1::i!0@1#3 =
  f::1::i!0@1#2 + 1
f::1::a!0@1#3 =
  store(f::1::a!0@1#2,
        f::1::i!0@1#3,
        f::1::i!0@1#3
f::1::i!0@1#4 =
  f::1::i!0@1#3 + 1
f::1::a!0@1#4 =
  store(f::1::a!0@1#3,
        f::1::i!0@1#4,
        f::1::i!0@1#4)
f::1::i!0@1#5 =
  f::1::i!0@1#4 + 1
assert !(f::1::i!0@1#5 < 3)
f::1::b!0@1#2 =
  select(f::1::a!0@1#4, 0) +
  select(f::1::a!0@1#4, 1)
```

Figure 4.6: Renaming arrays

store, receives an array, an index, and a value, and returns a new array with the value stored at that index updated with the new value. So, in the example the assignment `a[i] = i` is transformed into a new assignment with a store operation (`f::1::a!0@1#4 = store(...,...,...)`) and the array access `a[0]` is transformed into a select operation `select(f::1::a!0@1#4, 0)`.

Finally, observe the use of an `assert` statement to obtain a soundness result (check if executions requiring more than `K` iterations exist). We remark that, since arrays are modeled as applicative structures, the transformation into SA form does not introduce any novelties with respect to the previous sections.

4.7 Structures and Unions

The program in Figure 4.7 (left) shows the declaration of a struct type definition `X` with three fields, and a main function. The function declares a variable of type `X`, assigns some values to the fields of the struct, and returns the sum of the fields. The SA result may look a bit confusing, but it shows

how CBMC handles structs. Since we are asking for the encoding to be generated in the smt2 format, CBMC uses the tuples theory to encode structs. A tuple consists in an ordered set of elements `(A1 , ... , An)` where `Ai` is of some type supported by the SMT solver. Two operations are defined over tuples:

- **update**: receives a tuple `t`, a position `p` and a value `v` and returns a new tuple `t'` where every element except the one in position `p` has the same value as in `t`. The value of the element in position `p` is equal to `v`;

- **project**: receives a tuple `t` and a position `p` and returns the value in position `p` of the tuple.

```
typedef struct x{
  int a;
  char b;
  long int c;
} X;

int main(){
  X s;
  s.a = 1;
  s.b = 2;
  s.c = 3;
  int c = s.a + s.b
        + s.c;
  return c;
}
```

```
main::1::s!0@2#1 =
  update(3,0,main::1::s!0@2#0,1)
main::1::s!0@2#2 =
  update(3,1,main::1::s!0@2#1,2)
main::1::s!0@2#3 =
  update(3,2,main::1::s!0@2#2,3)
main::1::c!0@2#1 =
  project(3,0,main::1::s!0@2#3) +
  project(3,1,main::1::s!0@2#3) +
  project(3,2,main::1::s!0@2#3)
```

Figure 4.7: Renaming structs and struct's fields

Similarly to what happened with arrays, assignments to a field of a struct are now seen as applicative. An assignment is done through an `update` and the access to a field in an expression is done through `project`. The names of the fields are not used at all. So in the example, the assignment `s.b = 2` is transformed into `main::1::s!0@2#2 = update(3,1,main::1::s!0@2#1,2)`. Note the use of `update` and its parameters, where the first indicates the size of the tuple, the second indicates the index to be updated, the third indicates the tuple to be updated and the last indicates the new value.

Although unions are usually presented together with structures, in CBMC, at the SA level, they are treated in much the same way as for example an integer variable. The only difference is that the bitvector size may change along the time since the fields may have different sizes. The effect of this is simply the SA variables being declared with different sizes.

4.8 Pointers

The main goal of this section is to understand how CBMC deals with pointers and memory allocation. The main function of the program shown in Figure 4.8 uses an integer variable `a` and an integer pointer variable `d`. After some operations with these two variables, function `f` is called with the content from memory position `d` and the location of `a`.

```
#include<stdlib.h>

int f(int a, int *b){
  *b = a * 2;
  int c = *b + 100;
  return c;
}

int main(){
  int a = 10;
  int *d =
   malloc(sizeof(int));
  *d = 20;
  a = f(*d,&a);
  a = a * 2;
  return a;
}
```

```
main::1::a!0@2#1 = 10
main::$tmp::return_value_malloc$1!0@2#1 =
  nondet_symex::0
main::1::d!0@2#1 =
  main::$tmp::return_value_malloc$1!0@2#1
main::1::d$object!0#1 = 20
f::a!0@1#1 = main::1::d$object!0#1
main::1::a!0@2#2 = f::a!0@1#1 * 2
f::1::c!0@2#1 = main::1::a!0@2#2 + 100
main::1::a!0@2#3 = f::1::c!0@2#1
```

Figure 4.8: Renaming pointers

As we can see on the right part of the figure, a pointer is represented exactly in the same way as a variable. The only difference is when we are accessing the value in the address pointed by the pointer. We can see in the assignment `*d = 20` that a marker *object* is used, which corresponds to another SA variable. Note also the fact that the element `@` is not used when assigning a value to the pointer target. So the value after the `#` is always incremented independently from the function call. When there is a function

call with a parameter that is a pointer to a local variable, as in `f (...,&a)` of this example, the variable in the target function `a` is renamed to the variable in the source function. In our example, the input parameter `*b` gets renamed by the variable `a`, as shown by the assignment `c = *b + 100` which is transformed into `f::1::c!0@2#1 = main::1::a!0@2#2 + 100`.

4.9 Lessons Learned

With these experiences, we were able to realize how CBMC renames variables in order to transform a program into SA form. In the renaming of global variables an index is added to the name at each new assignment, counting the number of times the variable has been assigned so far. For local variables the function name in which it is declared is appended to the name, as well as two indexes: one counting the number of assignments made to that variable so far in a single function call, the other counting the number of function calls in the whole program. We were also able to see that function calls are inlined, and values must be propagated into and out of the function (sometimes requiring auxiliary variables). For arrays, and structures, we have seen that they are encoded as applicative, as explained in Sections 4.6 and 4.7 and consequently it becomes straightforward to convert to SA form instructions that manipulate such structures.

There exist some aspects of C, handled by CBMC, that are not required for SPARK. They include pointers and scopes. Even though they are not required in SPARK, we have analyzed them in order to complete this study. When using a pointer, multiple variables may be used to refer directly to the pointer and to the contents of the memory location pointed by it. The scopes are identified by appending a substring to the variable name at each level, which in turn contains a counter for the scopes declared previously at the same level. Finally, static variables are renamed using a substring which identifies in some way the file in which it is declared.

Chapter 5

SPARK-BMC

SPARK-BMC is a prototype bounded model checker for SPARK programs, that follows closely the steps described in Section 3.1. The tool checks *valid* SPARK programs for property violations as will be explained below. It assumes, without checking, that the input program passes the Examiner validity checks. The tool is being developed in Haskell and uses as backend the SMT-solver Z3 [dMB08]. SPARK-BMC is an open-source project[1].

We start this chapter by explaining some decisions that were made in the design of this tool. We later present implementation details about the steps required to create a bounded model of a SPARK program and send it to a solver. We finish the chapter with instructions on how to use the tool and present some examples of its use.

5.1 Implementation Choices

In this section, we give a brief description of the technologies being used to develop the BMC for SPARK programs. We also show the motivations behind these choices. Since the explanation of such technologies is outside the context from this thesis, we present them without giving great details. To have a full idea about the presented concepts the reader should refer to the references given throughout the chapter.

[1] available from the repository `https://bitbucket.org/vhaslab/spark-src`

5.1.1 Programming Language

The elected programming language for the implementation of SPARK-BMC was Haskell[2]. Haskell [OGS08] is a purely functional programming language, with polymorphic static typing, non-strict semantics and a monadic I/O system. Haskell provides a very advanced type system (which incorporates type classes and generalized algebraic data types) and very handy features such as pattern matching, list comprehensions, a module system and a rich set of libraries. The main motivations for choosing Haskell as a programming language in the development of SPARK-BMC are as follows:

- Productivity: since Haskell is a high level language, and due to its type system, it allows for a fast development process - less bugs, less time in debugging;
- Functional programming languages make easier the definition and manipulation of ASTs mainly due to the presence of algebraic data types and pattern matching - e.g. *uniplate*[3] makes very easy the application of generic transformations and queries to ASTs;
- Haskell is a very popular functional language at the moment, which makes it very easy to find information, support[4] and libraries[5];
- Even if Haskell is not the best language in terms of performance, the computational time in an application like SPARK-BMC is dominated by the SAT or SMT solver, so this performance aspect is not very relevant.

Beyond the facts presented above, the people involved in this project had experience with Haskell, which made the difference when comparing to other functional programming languages such OCaml.

[2] `http://www.haskell.org`
[3] `http://hackage.haskell.org/package/uniplate`
[4] `http://stackoverflow.com/questions/tagged/haskell`
[5] `http://hackage.haskell.org/`

5.1.2 Parser

Having chosen the development language, a parser for SPARK had to be created. This task has been carried out as part of this thesis and also of the activities of the AVIACC project with the collaboration of the whole team. The idea was to construct a robust parser that could be used not only for this project, but also for other tools developed in the context of AVIACC. The parser is currently an open source tool under the BSD 3 license, available from `https://bitbucket.org/vhaslab/spark-parser`.

Between the many available parser technologies for Haskell, we have chosen the monadic combinator parsing library Parsec[6]. The idea of parser combinators is to combine basic parsers with higher-order combinators to construct more complex parsers, allowing the parser to be build iteratively. The result of such combinators is a top-down recursive descent parser. Using such an approach has many advantages: the language used to create the parser is the same as the host language, thus, there is no need to learn a new language to use different tools to generate, or compile the parser and it is possible to take advantage of all the benefits of Haskell as for example as the type system or development environment.

5.1.3 Satisfiability Solver

The first decision when choosing a satisfiability solver was to decide if the main solver should be a SAT or SMT solver. The idea is that in the future the tool will allow the user to choose one solver from a set of supported solvers, both SAT and SMT. However, for this first version of the tool we had to give priority to one. According to particular tools described in [CFMS12] and [AMP09] the use of SMT solvers has many advantages, regarding for example compactness of the formulas (the size of the formulas does not depend on the bit-vector size neither on the array size), and scalability.

Our choice was to use Z3 [dMB08] as proof tool. Z3 is a high-performance SMT solver being developed at Microsoft Research. The code is open source and there are no restrictions for academic use. Moreover, open source bind-

[6] `http://legacy.cs.uu.nl/daan/parsec.html`

ings[7] are available for Haskell, which allow for the use of some convenient features, as well as a direct interaction with the solver.

The bindings for Z3 are available as an instance of *monad* which simplifies the development process. This is mainly due to the fact that a monad in Haskell is a computational container that allows to hide the 'bureaucratic' technical details of processing the data in steps and provides a simple interface for the programmer.

5.2 SPARK-BMC Internals

SPARK-BMC is a tool which performs Bounded Model Checking (BMC) to SPARK programs, following the steps described in Chapter 3. The tool takes as input a program written in SPARK, parses it, and creates an AST with the program representation. The program is then transformed, simplified, and normalized (by multiple traversals of the AST), and two sets of logical formulas are extracted, describing logically the operational contents of the program, and the properties to be established. Finally, the SMT solver is used to check if there exists a property violation; if so, a counter-example is shown.

SPARK-BMC checks for properties annotated in the code. Annotations are inserted as comments beginning with `--%`, distinct from SPARK annotations. This allows the user to verify different properties using different tools: a program annotated for SPARK-BMC is still a valid SPARK program that can be checked by the usual SPARK toolset. Our tool works with the following annotations:

```
--% assert C;
--% assume C;
--% notOverflow(op,type,e1,e2);
```

The two basic annotations `assert C` and `assume C`, with `C` a quantifier-free formula, are similar to those used in CBMC and described in Section 3.1. `assert C` means that `C` must be satisfied by the current state, so the condition `C` is true for all executions of the program. `assume C` restricts the

[7] `http://hackage.haskell.org/package/z3`

properties considered by allowing assume-guarantee reasoning. `assume C` indicates that condition `C` will be taken as true at the current state. If the condition `C` is false in a run of the program, then all the subsequent property checks of the program will vacuously pass. The `notOverflow` annotation is later translated into a set of asserts containing formulas that depend on the logical model used for numeric values.

The rest of this section presents the details of the several transformation steps on the original program, and the interaction with the SMT solver. The order in which the transformation are applied is the order they are presented. We use the program shown in Figure 2.1 as a running example. The code presented is copied from the tool output, when it runs in the debugging mode.

5.2.1 Program Simplification

The first step is to rewrite the input program into an equivalent one that uses only a restricted set of statements, namely, `loop`, `exit`, `if-then`, `if-then-else` and assignment. Table 5.1 shows the transformations applied to the input program. The first row of the table shows that `case` statements are translated into `if` statements. On the left-hand side, we show different uses of the **`when`** clause and show its transformation on the right-hand side where an `if` statement is used. The `elsif` clause, as in most programming languages is just syntactic sugar. Therefore, we expand it as shown in the table. Also, `exit` statements with the **`when`** clause are trivially transformed into `if` statements containing only an `exit` statement without a **`when`** clause.

In order to understand the transformations applied to `loop` statements, recall what has been said about loop statements in Section 2.1. In particular note that the most general form of iteration uses an infinite loop with an abrupt `exit` statement, and that in SPARK every `exit` statement must refer to the innermost enclosing loop. With this in mind and following the suggestion given in [Bar03], every loop with an iteration scheme can be converted into a loop with no iteration scheme. Table 5.1 shows the different iteration schemes and their conversion to a loop with no iteration scheme. For-loops can also contain the keyword `reverse` for the iteration to occur backwards.

`exit when c;`	`if c then`
	`  exit;`
	`end if;`

`case E is`
`  when A => S1;`
`  when B .. D => S2;`
`  when others => S3;`
`end case;`
	`if (E = A) then`
	`  S1;`
	`elsif (E in B .. D) then`
	`  S2;`
	`else`
	`  S3;`
	`end if;`

`if C1 then`
`  S1;`
`elsif C2 then`
`  S2;`
`[...]`
`else`
`  Sn;`
`end if;`
	`if C1 then`
	`  S1;`
	`else if C2 then`
	`  S2;`
	`  [...] else`
	`  Sn;`
	`  [...]`
	`  end if;`
	`end if;`

`while E loop`
`  S;`
`end loop;`
	`loop`
	`  exit when not E;`
	`  S;`
	`end loop;`

`for I in T range L..U loop`
`  S;`
`end loop;`
	`if L <= U then`
	`  I := L;`
	`  loop`
	`  S;`
	`  exit when I = U;`
	`  I := T'Succ(I);`
	`  end loop;`
	`end if;`

`for I in T loop`
`  S;`
`end loop;`
	`I := T'First;`
	`loop`
	`  S;`
	`  exit when I = T'Last;`
	`  I := T'Succ(I);`
	`end loop;`

Table 5.1: Transformations applied in the simplification step

In this case, the attributes must be changed in a trivial way to capture this semantics.

In our running example (Figure 2.1) only the `exit` statement with the `when` clause gets transformed. The result of this transformation may be seen in Figure 5.1. Note that we have removed the SPARK annotations to avoid confusion.

```
package body Marray is
  procedure MaxArray(V: in VArray; M: out Index)
  is
    I: Integer;
    Max: Index;
  begin
    Max := Index'First;
    I := Index'First+1;
    loop
      if (I > Index'Last) then
        exit;
      end if;
      if V(I) > V(Max) then
         Max := I;
      end if;
      I := I + 1;
    end loop;
    M := Max;
  end MaxArray;
end MArray;
```

Figure 5.1: After the simplification process

5.2.2 Program Instrumentation

Similarly to CBMC, the property annotations can be inserted by the user to express properties that should hold (very helpful for instance for debugging purposes), or else can be added automatically by an instrumentation tool that analyzes the program and inserts 'obvious' annotations. In SPARK code it is particularly useful to check statically for runtime exceptions; the SPARK Examiner certainly does this, but it requires annotating loop invariants in the code. These properties (in particular overflow, array out of bound access and division by zero) can be instrumented automatically, and with SPARK-BMC they can be checked without requiring loop invariants, as will be illustrated later.

Annotations inserted by instrumentation include a `notOverflow` annotation for each arithmetic operation that can possibly cause overflow; for array out of bounds accesses, it inserts an assert to check the validity of each index; division by zero is handled by inserting, for each arithmetic division, an `assert` to check that the denominator is different from zero. The example shown in Figure 5.1 would be annotated as shown in Figure 5.2 to check for overflow and array out of bound access.

```
package body Marray is
  procedure MaxArray(V: in VArray; M: out Index)
  is
    I: Integer;
    Max: Index;
  begin
    Max := Index'First;
    I := Index'First+1;
    loop
      if (I > Index'Last) then
        exit;
      end if;
      --% assert (I >= VARRAY'FIRST(1)) and (I <= VARRAY'LAST(1));
      --% assert (MAX >= VARRAY'FIRST(1)) and (MAX <= VARRAY'LAST(1));
      if V(I) > V(Max) then
         Max := I;
      end if;
      --% notOverflow(+,INTEGER,I,1);
      I := I + 1;
    end loop;
    M := Max;
  end MaxArray;
end MArray;
```

Figure 5.2: After the instrumentation process

5.2.3 Subprogram Inlining

The inlining of routine calls consists in taking the entry point provided by the user and recursively removing subprogram calls. For procedures and functions used as standalone statements, this is done by simply replacing the subprogram call by the respective body. For function calls occurring as part of expressions, the function body is inserted exactly before the statement which contains the call, and an auxiliary variable is used to propagate the return value. Auxiliary variables are also used to propagate the values of parameters inside the callee and back to the caller subprogram, taking into account their modes (`in`, `out` or `in out`). Since different contexts are being merged, identifiers are renamed to avoid conflicts, by adding as prefix the package and subprogram identifiers (this is also useful to keep information about the identifier's context). The result of this transformation step is a monolithic program with no calls.

During this process, type and subtype declarations, as well as variable declarations are collected as follows. For integer types and subtypes, we keep their lower and upper bounds; for enumerations types, the corresponding

literals (in the respective order); for enumerations subtypes the order of their first and last element as well as the type they extend; for modular types, the upper bound; for array types and subtypes, the types of their indexes and elements; finally for records, we keep their fields' names and type. Figure 5.3 shows the information kept for each type shown in Figure 2.3. For variables declarations, only the name and type are collected. The information collected during this process is kept and used by some of the follow transformations as will be shown.

```
Nat:         integer     - [0, 2147483647];
T:           mod         - 4;
Day:         enumeration - (Mon,Tue,Wed,Thur,Fri,Sat,Sun);
Counter:     integer     - [1, 10];
Weekday:     enumeration - [0,4] - Day;
Object:      record      - [(X: Integer), (Y: Integer)];
Circle:      record      - [(X: Integer), (Y: Integer), (Radius: Integer)];
Point:       record      - [(X: Integer), (Y: Integer)];
Tuple:       array       - [Integer] - [Day];
WorkHours:   array       - [Weekday] - [Nat];
Matrix:      array       - [Counter, Integer] - [Integer];
```

Figure 5.3: Information kept about the types

5.2.4 Eliminating Attributes and Enumeration Literals

SPARK attributes are a distinct feature of the language (and of course also Ada) which is not present in anyway in C. As such, it has to be addressed in a dedicated way: inspiration cannot be found in CBMC. SPARK attributes apply to types and subtypes. Since we assume that the program being verified is always a valid SPARK program, it is known beforehand that the enumeration literals are being used correctly. Moreover, SPARK forbids the overloading of enumeration literals. With this in mind, we translate enumeration literals into integer values (the respective position in the enumeration, starting with 0 for the first) and get rid of attributes, replacing them by equivalent expressions. After this, enumerations are manipulated as integer values. The only exception is for literals of Boolean type. Despite SPARK

Booleans being defined as enumeration types, they have a special semantics because they can be used in Boolean expressions.

rmAttr(l) = getPos(l)
rmAttr(N1 op N2) = rmAttr(N1) *op* rmAttr(N2)
rmAttr(S'First) = lower(S)
rmAttr(S'Last) = upper(S)
rmAttr(A'First) = lower (getIndType(1,A))
rmAttr(A'Last) = upper (getIndType(1,A))
rmAttr(A'First(N)) = lower (getIndType(rmAttr(N),A))
rmAttr(A'Last(N)) = upper (getIndType(rmAttr(N),A))
rmAttr(A'Length) = rmAttr(A'Last - A'First + 1)
rmAttr(A'Length(N)) = rmAttr(A'Last(N) - A'First(N) + 1)
rmAttr(S'Min(N1,N2)) = (rmAttr(N1) $\leq$ rmAttr(N2)) ? rmAttr(N1) : rmAttr(N2)
rmAttr(S'Max(N1,N2)) = (rmAttr(N1) $\geq$ rmAttr(N2)) ? rmAttr(N1) : rmAttr(N2)
rmAttr(S'Pos(N)) = getPos(rmAttr(N))
rmAttr(S'Val(N)) = rmAttr(N)
rmAttr(S'Pred(N)) = rmAttr(N) - 1
rmAttr(S'Succ(N)) = rmAttr(N) + 1
rmAttr(B'First) = False
rmAttr(B'Last) = True
...

Figure 5.4: Attribute removal function

Having the types and variable declarations obtained in the previous step, it is now possible to convert enumeration elements into integers, and attributes into equivalent expressions using only integer values. Replacing enumeration elements by integers is straightforward; it suffices to replace the element by its position in the enumeration. Figure 5.4 shows a recursive function to remove attributes and enumerations from an expression. A denotes an object of an array type, S denotes an object of integer or enumeration type or subtype (not Boolean), B denotes an object of Boolean type; N denotes an expression and l an enumeration literal. Function lower (resp. upper) gives the lower (resp. upper) bound of an integer or enumeration type/subtype; getIndType receives a positive integer (indicating one dimension of the array) and an array type and returns its type; getPos returns the position of a literal in an enumeration. Other cases are analogous to the ones shown.

```
ARRAY_SIZE := 10;
MAX := 1;
--% notoverflow(+,INTEGER,1,1);
I := (1 + 1);
loop
  if (I > ARRAY_SIZE) then
    exit;
  end if;
  --% assert (I >= 1) and (I <= ARRAY_SIZE);
  --% assert (MAX >= 1) and (MAX <= ARRAY_SIZE);
  if (V(I) > V(MAX)) then
    MAX := I;
  end if;
  --% notoverflow(+,INTEGER,I,1);
  I := (I + 1);
end loop;
M := MAX;
```

Figure 5.5: Transforming enumerations and attributes into integer expressions

5.2.5 Loop Unwinding

Loops can at this stage be seen as blocks of code containing backward-goto and forward-goto statements. Since the iteration schemes have been removed during the simplification step, only primitive loops are present at this stage, and the only way of exiting such a loop is through an explicit `exit`, equivalent to a forward-goto statement. On the other hand, reaching the end of a loop block produces a backward-goto to the beginning of the loop.

In the present step, in order to produce a bounded model, each loop gets unwound a fixed number of times K. Loop headers are removed, and loop bodies are replicated K times. However, `exit` statements complicate this picture: the statements belonging to the loop body following a reached exit statement must not be executed. Since this intermediate representation of the program will not itself be executed, neither it is used directly to build a logical encoding, we introduce an artificial loop unwind wrapper for the code belonging to the unwound loop. Within this wrapper, reaching an exit statement means that none of the subsequent statements inside the wrapper should get executed. Such wrappers have other uses as will be seen below. They will be removed in the final normalization step, as will be the `exit` statements.

Depending on the user's choice, an unwinding assertion can be placed at the end of the expanded code (inside the loop wrapper) with the aim of ensuring that the loop has been sufficiently expanded. This process is the same as the one explained in Section 3.1, more precisely in the discussion about creating a bounded model. Remember that the idea of `assert False` is that, whenever that point is reached in a certain path, such path requires more than `K` iterations and so there is an assert violation. If the user chooses to turn off property validation in paths beyond the limit established by the bound for loop iterations, the unwinding assumption (`assume False`) is placed instead. Remember that whenever the point in which the unwinding assumption was placed is reached by a certain execution, all subsequent assertions in this execution will be trivially discharged.

```
ARRAY_SIZE := 10;
MAX := 1;
--% notoverflow(+,INTEGER,1,1);
I := (1 + 1);
LOOP_WRAPPER
  if (I > ARRAY_SIZE) then
    exit;
  end if;
  --% assert (I >= 1) and (I <= ARRAY_SIZE);
  --% assert (MAX >= 1) and (MAX <= ARRAY_SIZE);
  if (V(I) > V(MAX)) then
    MAX := I;
  end if;
  --% notoverflow(+,INTEGER,I,1);
  I := (I + 1);
  if (I > ARRAY_SIZE) then
    exit;
  end if;
  --% assert (I >= 1) and (I <= ARRAY_SIZE);
  --% assert (MAX >= 1) and (MAX <= ARRAY_SIZE);
  if (V(I) > V(MAX)) then
    MAX := I;
  end if;
  --% notoverflow(+,INTEGER,I,1);
  I := (I + 1);
  --% assert False;
END_LOOP_WRAPPER;
M := MAX;
```

Figure 5.6: Loop unwinding 2x

Figure 5.6 shows our running example, with the initial loop unwound twice. Note in particular the code standing between `LOOP_WRAPPER` and `END_LOOP_WRAPPER`, which is exactly the loop body replicated twice. Note

also that for this example an unwinding assertion was inserted. If an unwinding assumption was chosen, the only difference would be the line containing the unwinding assertion (one would have assume instead of assert). Finally, note that the code standing between the second `exit` statement and the unwinding assertion is meaningless, since whenever it gets executed, the `assert False` also gets executed.

5.2.6 Single-Assignment Transformation

A crucial step towards the normalization of the program is its transformation into an Single Assignment (SA) form. In this transformation, multiple indexed versions of each variable are used – a new version is introduced for each assignment to the original variable, so that in every execution path, once a variable has been read or assigned it will not be assigned again.

While the transformation of straight-line code is quite straightforward (it suffices to index each variable with the number of times it has been assigned so far), code with multiple branches poses some challenges. Recall that different variables will be used to represent the same variable of the original program in different paths of the SA program, and these have to be merged when these paths meet. The only statement left with multiple branches at this stage of the transformation workflow is the `if` statement. Loop wrappers may also have multiple branches, depending on the exit statements; they will be discussed in the next subsection.

Returning to the `if` statements, the two final versions of each variable (one from each branch) must be merged; this is achieved by inserting, immediately after the conditional, an assignment to each modified variable, making use of *conditional expressions*, à la C. These expressions are not present in the SPARK programming language, and even though variables could alternatively be merged by assigning the same variable in both branches, we have chosen to include conditional expressions in the AST for the sake of compactness and readability. What is more, these expressions can be preserved at the logical level since they are present in the SMT logic.

Our conversion algorithm traverses the AST and appends an appropriate

index to each variable occurrence. To deal with multiple branch statements, the algorithm makes use of two counters for each variable; one family of counters (`R`) keeps the versions that should be used when a variable is read, the other (`W`) keeps the last versions of the variables that have been used for writing. Both `R(v)` and `W(v)` are incremented when variable `v` is assigned. However, when entering an *else* branch `R(v)` must be reset to the value it had immediately before entering the conditional. At the merge point, the values of `W` at the end of both branches are used for inserting an assignment with the appropriate conditional expression.

```
if (X = Y) then
  X := X + Y;
  Y := Y + 1;
else
  X := X + Y;
end if;
```

```
                                    -- R = {(X,2), (Y,4)}; W = {(X,3),(Y,4)}
if (X#2 = Y#4) then
  X#4 := X#2 + Y#4;                 -- R = {(X,4), (Y,4)}; W = {(X,4),(Y,4)}
  Y#5 := Y#4 + 1;                   -- R = {(X,4), (Y,5)}; W = {(X,4),(Y,5)}
else
  X#5 := X#2 + Y#4;                 -- R = {(X,5), (Y,4)}; W = {(X,5),(Y,5)}
end if;
X#6 := (X#2 = Y#4) ? X#4 : X#5;
Y#6 := (X#2 = Y#4) ? Y#5 : Y#4;
                                    -- R = {(X,6), (Y,6)}; W = {(X,6),(Y,6)}
```

Figure 5.7: Example of conversion to SA

Consider the example in Figure 5.7. As comments it is possible to observe the evolution of the counters `R` and `W`. The fact that `R(X)` and `W(X)` are different indicates that this is part of some *else* branch. Observe how the values of `R(X)` and `R(Y)` before entering the conditional are used in both branches when `X` and `Y` are read.

Special attention must be given to assignments involving arrays. For the purpose of obtaining a logical encoding, arrays are seen as *applicative*: each array corresponds to a single variable, updated through a *store* function. So an assignment of the form `A(X) := Y` is first transformed into `A := store(A,X,Y)`, and conversion to SA form then produces the instruction `A2 := store(A1,X',Y')`, where `X'` and `Y'` correspond to `X` and `Y`, with

possible renaming of variables due to transformation to SA form.

```
ARRAY_SIZE#1 := 10;
MAX#2 := 1;
--% notOverflow(+,INTEGER,1,1);
I#2 := (1 + 1);
LOOP_WRAPPER
   if (I#2 > ARRAY_SIZE#1) then
     exit;                                    -- [(I,2),(M,1),(MAX,2),(V,1)]
   end if;
   --% assert (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1);
   --% assert (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1);
   if (V#1(I#2) > V#1(MAX#2)) then
     MAX#3 := I#2;
   end if;
   MAX#4 := (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2;
   --% notOverflow(+,INTEGER,I#2,1);
   I#3 := (I#2 + 1);
   if (I#3 > ARRAY_SIZE#1) then
     exit;                                    -- [(I,3),(M,1),(MAX,4),(V,1)]
   end if;
   --% assert (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1);
   --% assert (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1);
   if (V#1(I#3) > V#1(MAX#4)) then
     MAX#5 := I#3;
   end if;
   MAX#6 := (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4;
   --% notOverflow(+,INTEGER,I#3,1);
   I#4 := (I#3 + 1);
   --% assert False;
END_LOOP_WRAPPER;                           -- [(I,5),(MAX,7)]
M#2 := MAX#7;
```

Figure 5.8: Single assignment program representation

Figure 5.8 presents the state of our running example after conversion to SA form. The comments after the exit statements and the `END_LOOP_WRAPPER` keyword may be ignored for now. We will refer to them when dealing with the exit statement removal, below. Note in particular that `MAX#7` is not assigned during this step because its value depends on the `exit` statement reached. Such assignments are added in the next step, when removing `exit` statements.

5.2.7 Conditional Normal Form Normalization

Conditional Normal Form (CNF) normalization transforms the program into a sequence of statements of the form `if b then S`, where `S` must be an *assignment*, *assert* or *assume* instruction. Before such a form is reached,

`exit` statements must be removed. For that we divide this step in two parts. First, we start by normalizing the program in the form just described, but allowing `S` to be an `exit` statement. Later we remove these statements.

For the first part, `if` statements with `else` branch are rewritten into a sequence of two `if` statements with mutually exclusive conditions. This transformation is correct since in SA form the value of the boolean expression cannot possibly be modified by subsequent instructions. Nested `if` statements are then pushed up in the AST, by traversing it and collecting the necessary path conditions for reaching each assignment, `exit`, `assert` or `assume` instruction, and then creating an `if` statement with the conjunction of the collected conditions. Figure 5.9 shows the normalization of the example shown in Figure 5.7. Figure 5.10 (top) shows the result of performing these operations on our running example.

```
if (X#2 = Y#4)          then X#4 := X#2 + Y#4;                    end if;
if (X#2 = Y#4)          then Y#5 := Y#4 + 1;                      end if;
if (not (X#2 = Y#4))    then X#5 := X#2 + Y#4;                    end if;
if (True)               then X#6 := (X#2 = Y#4) ? X#4 : X#5;      end if;
if (True)               then Y#6 := (X#2 = Y#4) ? Y#5 : Y#4;      end if;
```

Figure 5.9: Example of normalization into conditional normal form

For the exit statements removal, recall that when such a statement is reached, this means that none of the following instructions in the corresponding loop wrapper should be reached. Moreover, the values of the variables after the loop wrapper must be in accordance with the first exit statement reached. If no exit statement is reached then an assert or an assume annotation previously inserted must be reached, as explained before. Note that the preparation for this step starts when the code is converted into SA form: for each exit statement, the current version of each variable at that point was kept, and at the end of each loop wrapper, a new version for each modified variable was reserved and kept together with the wrapper. This version is the one that is used immediately after the loop wrapper: its value reflects the exit statement that has been reached. This information is displayed as comments in the example shown in Figure 5.8.

At this stage each exit statement is guarded by a condition `C`; the state-

ment is reached if and only if `C` evaluates to true. To ensure that none of the subsequent statements (including other exit statements and the unwinding annotation) are reached after an exit statement is, it is enough to ensure that none of the subsequent guards evaluate to true, by propagating the condition `not C` through them. In order for variables to hold the correct values after the loop wrapper, assignments to the reserved variables are inserted using the current variables that were kept together with the exit statement. The guards for these assignments are the same as the exit guard. Loop wrappers and all the information kept for this step may now be removed.

After performing these updates on the AST, our running example is as shown in Figure 5.10 (bottom). Note the use of auxiliary variables to keep the loop conditions. These variables are only used for keeping the program easy to understand and may be removed at any instance, by propagating their values.

5.2.8 Creating the Logical Encoding

There are no language-specific aspects in the next step. After all the previous transformations, the program is now a sequence of statements of the form `if C then S`, where `S` may only be an assignment or an instruction with the form of an annotation (`assert`, `notOverflow`, or `assume`). As described in Section 3.1, one now has to extract the two sets of formulas $\mathcal{C}$ and $\mathcal{P}$. This is straightforward, by traversing the list of statements, translating `if` statements into implications and assignments into equalities. The formulas with assert and assume statements (including those in the form `notOverflow`) are collected in $\mathcal{P}$, and the remaining implications in $\mathcal{C}$. Figure 5.11 shows these two sets of expressions for our running example.

5.2.9 Solver Interaction

Now that we have $\mathcal{C}$ and $\mathcal{P}$ the satisfiability of $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ modulo a background theory has to be checked. The simultaneous presence in the language of discrete types with modular semantics and of signed integers for which runtime overflow exceptions are raised led us to elect fixed-size bit-vectors

```
True => ARRAY_SIZE#1 := 10;
True => MAX#2 := 1;
True => --% notoverflow(+,INTEGER,1,1);
True => I#2 := (1 + 1);
LOOP_WRAPPER
   (I#2 > ARRAY_SIZE#1) => exit; -- [(I,2),(M,1),(MAX,2),(V,1)]
   True => --% assert (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1);
   True => --% assert (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1);
   (V#1(I#2) > V#1(MAX#2)) => MAX#3 := I#2;
   True => MAX#4 := (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2;
   True => --% notoverflow(+,INTEGER,I#2,1);
   True => I#3 := (I#2 + 1);
   (I#3 > ARRAY_SIZE#1) => exit; -- [(I,3),(M,1),(MAX,4),(V,1)]
   True => --% assert (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1);
   True => --% assert (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1);
   (V#1(I#3) > V#1(MAX#4)) => MAX#5 := I#3;
   True => MAX#6 := (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4;
   True => --% notoverflow(+,INTEGER,I#3,1);
   True => I#4 := (I#3 + 1);
   True => --% assert False;
END_LOOP_WRAPPER; -- [(I,5),(MAX,7)]
True => M#2 := MAX#7;
```

```
True => ARRAY_SIZE#1 := 10;
True => MAX#2 := 1;
True => --% notoverflow(+,INTEGER,1,1);
True => I#2 := (1 + 1);
True => _NExit#1 := not((I#2 > ARRAY_SIZE#1));
(I#2 > ARRAY_SIZE#1) => MAX#7 := MAX#2;
(I#2 > ARRAY_SIZE#1) => I#5 := I#2;
_NExit#1 => --% assert (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1);
_NExit#1 => --% assert (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1);
_NExit#1 and (V#1(I#2) > V#1(MAX#2)) => MAX#3 := I#2;
_NExit#1 => MAX#4 := (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2;
_NExit#1 => --% notoverflow(+,INTEGER,I#2,1);
_NExit#1 => I#3 := (I#2 + 1);
True => _NExit#2 := not((I#3 > ARRAY_SIZE#1)) and _NExit#1;
_NExit#1 and (I#3 > ARRAY_SIZE#1) => MAX#7 := MAX#4;
_NExit#1 and (I#3 > ARRAY_SIZE#1) => I#5 := I#3;
_NExit#2 => --% assert (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1);
_NExit#2 => --% assert (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1);
_NExit#2 and (V#1(I#3) > V#1(MAX#4)) => MAX#5 := I#3;
_NExit#2 => MAX#6 := (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4;
_NExit#2 => --% notoverflow(+,INTEGER,I#3,1);
_NExit#2 => I#4 := (I#3 + 1);
_NExit#2 => --% assert False;
True => M#2 := MAX#7;
```

Figure 5.10: Conditional normal form normalization

for our primary encoding of discrete types: the modular semantics is directly captured by bit-vectors, and for signed integers, since overflow is protected by instrumented `notOverflow` annotations, it is indifferent to use bit-vectors or an unbounded integers encoding. Enumerations (converted to integers)

```
True -> (ARRAY_SIZE#1 = 10)
True -> (MAX#2 = 1)
True -> (I#2 = (1 + 1))
True -> (_NExit#1 = not((I#2 > ARRAY_SIZE#1)))
(I#2 > ARRAY_SIZE#1) -> (MAX#7 = MAX#2)
(I#2 > ARRAY_SIZE#1) -> (I#5 = I#2)
_NExit#1 and (V#1(I#2) > V#1(MAX#2)) -> (MAX#3 = I#2)
_NExit#1 -> (MAX#4 = (V#1(I#2) > V#1(MAX#2)) ? MAX#3 : MAX#2)
_NExit#1 -> (I#3 = (I#2 + 1))
True -> (_NExit#2 = not((I#3 > ARRAY_SIZE#1)) and _NExit#1)
_NExit#1 and (I#3 > ARRAY_SIZE#1) -> (MAX#7 = MAX#4)
_NExit#1 and (I#3 > ARRAY_SIZE#1) -> (I#5 = I#3)
_NExit#2 and (V#1(I#3) > V#1(MAX#4)) -> (MAX#5 = I#3)
_NExit#2 -> (MAX#6 = (V#1(I#3) > V#1(MAX#4)) ? MAX#5 : MAX#4)
_NExit#2 -> (I#4 = (I#3 + 1))
True -> (M#2 = MAX#7)
```

```
True -> notOverflow(+,INTEGER,1,1)
_NExit#1 -> (I#2 >= 1) and (I#2 <= ARRAY_SIZE#1)
_NExit#1 -> (MAX#2 >= 1) and (MAX#2 <= ARRAY_SIZE#1)
_NExit#1 -> notOverflow(+,INTEGER,I#2,1)
_NExit#2 -> (I#3 >= 1) and (I#3 <= ARRAY_SIZE#1)
_NExit#2 -> (MAX#4 >= 1) and (MAX#4 <= ARRAY_SIZE#1)
_NExit#2 -> notOverflow(+,INTEGER,I#3,1)
_NExit#2 -> False;
```

Figure 5.11: Two lists of formulas: $\mathcal{C}$ (top) and $\mathcal{P}$ (bottom)

are also encoded as bit-vectors.

As it was said before, the interaction with Z3 is done with bindings, and so, the communication with the solver is done iteratively recurring to monadic operators[8]. The first task is to create the adequate Z3 sorts for each SPARK predefined or user-defined type. Z3 bindings provides several functions to create different sorts as for example booleans, bit-vectors and arrays. For instance, to create a 32 bit integer, the function to create a bit-vector sort must be invoked with the desired size (32 in this case). A relation between these two classes of types, that is, between the SPARK types and Z3 sorts, is kept for later use when creating constants and expressions. Since Z3 does not support multiple-index arrays, SPARK arrays with multiple indexes are represented using nested arrays in Z3. The first line of Figure 5.12 shows an example of an array declaration in SPARK and, as comment, it shows the result of applying our strategy. The text shown as comment is written

[8] `http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Monad.html`

in the SMT-LIB format to illustrate how they would look if such format was used. Note that we do not generate any kind of formulas in the SMT-LIB format, however, in this case it may be helpful to understand how we translate multiple-indexed arrays into nested arrays.

```
type Matrix is array (A,B) of C -- (define-sort (Matrix)
                                --                ()
                                --                (Array A (Array B C))
                                -- )
--
X : Matrix;                     -- (declare-fun X () Matrix)
--
Y := ... X[a,b] ...;            -- ... (select (select X a) b) ...
--
X[a,b] := Y;                    -- (store X a (update (select X a) b Y))
```

Figure 5.12: Multiple indexed arrays to nested arrays

After having the sorts created, one may then create one Z3 constant for each program variable (one for each SA version). Remember that we have collected the variables declarations in the program, and we have a relation between SPARK types and Z3 sorts. Therefore, the construction of constants is done trivially using this information.

Formulas from $\mathcal{C}$ are interpreted and taken as assertions one by one, while formulas from $\mathcal{P}$ are used to build the expression $\neg \bigwedge \mathcal{P}$ which is then also taken as an assertion. The interpretation of SPARK expressions and creation of Z3 formulas is done in a programmatic way, by using several functions available in the Z3 bindings. Variables are represented by the corresponding constant, and operators are represented by the corresponding bit-vector operator.

After all expressions are defined in Z3, we check for satisfiability. The Z3 bindings provide a function for this, which returns a SAT or UNSAT result, and in the former case also a model that will be interpreted as a counter-example. In the case it is UNSAT, we also show the failing assertion. This is done by checking if the formulas from $\mathcal{P}$ are valid in the context of the model returned by the solver.

As an example of interaction with Z3 through bindings, consider Figure 5.13. On the top left corner, there is a variable declaration (assigned

```
...
X : Integer := 10;
...
--% assert X <= 10;
...
X := 20;
...
```

```
-- C
X1 = 10;
X2 = 20;
-- not P
not(X1 <= 10);
```

```
simple :: Z3 String
simple =
  do -- Creating Sorts
     int32Sort <- mkBvSort 32
     -- Creating Constants
     constX1 <- flip mkConst int32Sort =<< mkStringSymbol "X1"
     constX2 <- flip mkConst int32Sort =<< mkStringSymbol "X2"
     const10 <- mkInt 10 >>= mkInt2bv 32
     const20 <- mkInt 20 >>= mkInt2bv 32
     -- Creating and sending expressions from "C"
     assertCnstr =<< mkEq constX1 const10
     assertCnstr =<< mkEq constX2 const20
     -- Creating and sending expression "not P"
     assertCnstr =<< mkNot =<< mkBvsle constX1 const10
     -- Checking for Satisfiability
     (r,m) <- getModel
     case m of
       -- If it is SAT a counter example is returned
       Just m' -> showModel m'
       -- If it UNSAT the model is correct
       Nothing -> return $ show r
```

Figure 5.13: Example of interaction with Z3 through Haskell monadic bindings

with the number 10), an assert specification and another assignment with the number 20. The set of formulas $\mathcal{C}$ and $\mathcal{P}$ obtained after all the transformations are shown on the right hand side of the figure. An Haskell program, that verifies the satisfiability of this set of formulas $\mathcal{C} \cup \{\neg \bigwedge \mathcal{P}\}$ is shown on the bottom of the figure. It starts by declaring a sort to represent the 32 bit integer type. After that it declares the constants to represent the variables `X1` and `X2` as well as two constants to represent the value `10` and `20`. The first assignment is written as a Z3 expression and it is taken as an assertion and the same happens for the second assignment. For the assertion, it first creates a Z3 expression with the negation of the condition and then sets it as an assert. Once having the model encoded in Z3, the function `getModel` is used to check the satisfiability of the set of the defined assertions. The function returns a tuple, where the first element indicates if the model is

satisfiable or not and the second element contains a valuation for the model (in cases it is satisfiable).

5.3 Using SPARK-BMC

This section is devoted for the use of SPARK-BMC. For instructions about how to install the tool, see Appendix A. We start by describing the interactions with the tool, showing its main characteristics. The options available are presented and discussed, and examples of its use are given.

5.3.1 Basic Usage

Knowing what is behind our tool is now time to run the tool and to show how it can be used. If one executes the tool with no parameters the output is as shown in Figure 5.14.

```
$./spark-bmc
Usage:
 spark-bmc [OPTION...] file_name

  -v, -?                  --version                       show version number
  -h                      --help                          show help
  -e entry                --entry=entry                   entry point subprogram
  -u {assert,assume}      --unwind={assert,assume}
unwinding assertions or unwinding assumptions
  -b bound                --bound=bound                   unwinding bound
  -s {bit-vec, nosolver}  --solver={bit-vec, nosolver}    bit-vector encoding or no solver
```

Figure 5.14: Running SPARK-BMC with no parameters

The output shown in the figure indicates the arguments that should be given to the SPARK-BMC when invoking it. Argument `entry` is used to specify the program entry point. It must be a subprogram name that must exist in the file given as argument (`file_name`). The `solver` argument, is used to specify which theory should be used by the solver, if any. The only possible theory available at the moment of writing this thesis was bit-vectors. It is expected that in a near future, also unbounded integers will be available. If `nosolver` is chosen, then the formula is not even sent to the solver, however, the transformations to the initial program are written in a debug file called `spark-bmc-debug`. The argument `bound` is used to specify

the number of times each loop is unwound. At last, `unwind` sets whether an unwinding assertion or unwinding assumption is placed after the unwound loops. Next we show some examples of the use of SPARK-BMC.

5.3.2 Example: Maximum Element in Array

Here we run SPARK-BMC with some examples and present the results obtained. We start with our running example from Section 5.2 as shown in Figure 2.1. First of all, let us check this example with a bound of 2, and placing unwinding assumption for the loop unwinding. Figure 5.15 shows the result of performing such operation.

```
$./spark-bmc -s bit-vector -b 2 -u assume -e MaxArray example1.adb

UNSAT - that means the model is correct for the given bound

See spark-bmc-debug.adb for debugging purposes
```

Figure 5.15: Running SPARK-BMC with a bound of 2 and using an unwinding assumption

From the output it is possible to observe that the program is correct for executions requiring at most 2 iterations. The result of running the same example with the same bound, but using an unwinding assertion instead of an unwinding assumption is shown in Figure 5.16.

```
$./spark-bmc -s bit-vector -b 2 -u assert -e MaxArray example1.adb

See spark-bmc-debug.adb for debugging purposes

Unwinding Assertion Violation:
File: example1.adb
Line: 9
```

Figure 5.16: Running SPARK-BMC with a bound of 2 and using an unwinding assertion

Using an unwinding assertion, SPARK-BMC reports that there is a property violation, more precisely an unwinding assertion violation. From the

output it is possible to identify the loop that requires more iterations. Following the idea of BMC of software, if one cannot find bugs with a bound of `K`, nor prove the correctness of the program, then `K` must be incremented at least until reaching the limit of resources. Now let us run exactly the same example but with a bound of 10 as shown in Figure 5.17.

```
$./spark-bmc -s bit-vector -b 10 -u assert -e MaxArray example1.adb

UNSAT - that means the model is correct
```

Figure 5.17: SPARK-BMC, bound of 10 and using an unwinding assertion

With a bound of 10 it is already possible to prove that our program respects all properties being checked. Note that if the option **`assume`** was chosen instead (for the unwinding assumption to be used in the loop unwinding process), the tool would only say that the program is correct for executions requiring less than 10 iterations. No check of the fact that the loop was fully unrolled would be performed.

```
package body Marray is
  procedure MaxArray(V: in VArray; M: out Index)
  is
    I: Integer;
    Max: Index;
  begin
    Max := Index'First;
    I := Index'First+1;
    loop
      exit when I > Index'Last + 1;
      --# assert (for all J in Index range Index'First..(I-1)
      --#           => (V(J) <= V(Max)));
      --# assert (I >= Index'First) and (I <= Index'Last + 1);
      if V(I) > V(Max) then
         Max := I;
      end if;
      I := I + 1;
    end loop;
    M := Max;
  end MaxArray;
end MArray;
```

Figure 5.18: SPARK program with a bug

Let us now modify our example, introducing a discrete bug (in the sense it could be easily missed), to show how the tool can discover subtle bugs without

the need for user annotations. One common error would be to write the `exit` condition as shown in Figure 5.18. Actually this is the only modification to the initial program. It is easy to see that this alternative condition would cause an array out of bounds exception in the array access contained in the expression `V(I) > V(MAX)`, but such a bug can be easily missed. Our tool detects it automatically as shown in Figure 5.19.

```
$./spark-bmc -s bit-vector -b 10 -u assert -e MaxArray example1.adb

See spark-bmc-debug.adb for debugging purposes

Assertion Violation:

assert (I >= INDICES'FIRST) and (I <= INDICES'LAST)

File: SBMC_EXAMPLES/EXAMPLE1.ADB
Line: 82


For:
I#11 <- 11
ARRAY_SIZE#1 <- 10
```

Figure 5.19: SPARK-BMC finding bugs related to array out of bounds automatically

The SPARK tools (based on deductive verification) would generate a verification condition (labelled as `assert`) stating that the loop invariant is preserved by iterations of the loop, and another VC (labelled as `rtc check`) to enforce that whenever `V(I) > V(MAX)` is evaluated the value of `I` lies within the range of the array. For the code of Figure 2.1 both VCs are successfully discharged: no out-of-bounds access takes place. But if the `exit` condition is modified as shown in Figure 5.18, then the invariant preservation condition can no longer be proved (it fails in the last iteration). The `rtc check` is still proved, because it is a consequence of the invariant. If the invariant is corrected to `I <= Index'Last + 2`, then the invariant preservation VC is discharged, but not the `rtc check` – the invariant is now correct, but it does not prevent the out-of-bounds access. This example illustrates that with deductive verification it can be hard to detect exactly what went wrong – is the program unsafe, or is the user-provided invariant wrong?

5.3.3 Example: Factorial

Let us now check the factorial example first shown in Figure 2.7. In this example we will only check for the absence of operations that can cause overflow. The instrumentation of the program to check such properties is shown in Figure 5.20. To avoid confusion, we have removed the SPARK annotations.

```
package body Factorial is
  function fact(X: Integer) return Integer
  is
    F, I: Integer;
  begin
    F := 1;
    I := 1;
    loop
      --% notOverflow(+,Integer,X,1);
      exit when ( I = X + 1 );
      --% notOverflow(*,Integer,F,I);
      F := F * I;
      --% notOverflow(+,Integer,I,1);
      I := I + 1;
    end loop;
    return F;
  end fact;
end Factorial;
```

Figure 5.20: Factorial example with instrumentation for overflow analysis

For this example we are going to start by unwinding the loops only once. We start by checking the example using unwinding assumptions as shown in Figure 5.21.

```
$./spark-bmc -s bit-vector -b 1 -u assume -e fact example1.adb

See spark-bmc-debug.adb for debugging purposes

Assertion Violation:
notOverflow( + ,INTEGER,X,1)

File: SBMC_EXAMPLES/EXAMPLE1.ADB
Line: 111

For:
X#1 <- 2147483647
```

Figure 5.21: SPARK-BMC finding bugs related to overflow automatically

As shown above, SPARK-BMC easily finds a counter-example and shows

a value for the input argument which violates one property, more precisely the property `notOverflow( + ,INTEGER,X,1)`. This is a property that checks if an overflow occurs on the operation `I := X + 1`. Of course that when the value of `X` is equal to the maximum integer value, an overflow occurs. Actually in a 32 bit machine, it is not possible to calculate the factorial of a number greater than 12. However, one may still wan to check if others assertions fails. For that let us introduce an assume annotation in the code, stating that the input argument is greater than 0 and smaller than 13. Such an annotation may be seen as a pre-condition, which will limit the executions to be explored. Of course, functions invoking `fact` must guarantee that the given values are in such a range. The code with the assumption is shown in Figure 5.22.

```
package body Factorial is
  function fact(X: Integer) return Integer
  is
    F, I: Integer;
  begin
    --% assume X > 0 and X < 13;
    F := 1;
    I := 1;
    loop
      --% notOverflow(+,Integer,X,1);
      exit when ( I = X + 1 );
      --% notOverflow(*,Integer,F,I);
      F := F * I;
      --% notOverflow(+,Integer,I,1);
      I := I + 1;
    end loop;
    return F;
  end fact;
end Factorial;
```

Figure 5.22: Factorial example with assume annotation

Running now SPARK-BMC with a bound of 13 and an unwinding assertion it is possible to prove that the program is free of overflows whenever the input argument is greater than `0` and smaller than `13`.

Chapter 6

Conclusions

6.1 Discussion of Contributions

The main goal of this thesis was the development of a BMC for SPARK programs, which was successfully attained. Although the development of SPARK-BMC is still a work in progress (some features of SPARK are not yet covered), the tool's workflow is entirely implemented, and we are able to successfully check programs manipulating arrays and discrete types.

The process of building a parser for a full language is generally complex, however, in the case of SPARK it was even more complicated due to the lack of documentation. The existing book about the language dates from 2003 and is sometimes not clear about some functionalities of the language. There exists also official documentation[1] provided by Altran, however, the available language grammar is complex since it is adapted from Ada language. With the help of some additional empirical studies we were able to create a complete parser for the language which we see as a contribution of independent value. The parser is capable of parsing a SPARK program (specification and body) and creating an AST which is then used for the BMC transformations. Since the parser is an open source project, we now count on the community interested in using our parser to improve certain functionalities and to alert us about bugs that may exist. A parser functionality that would improve

[1] `http://docs.adacore.com/sparkdocs-docs/SPARK_LRM.htm`

directly SPARK-BMC is the parsing of configuration files (files in which it is possible to define the range of the default integers). At the current stage, we are using a constant value, however, the whole application is prepared to incorporate such parser improvement.

The program simplification process implemented by the tool is working as presented. Since the application of most transformation steps of the BMC method produces a valid and semantically equivalent program, they may be used in other contexts in order to normalize programs by reducing the number of different statements. Another functionality that may be reused is the transformation of enumerations and attributes into integer expressions. This is working as presented in Figure 5.4, and may be useful for the creation of other tools avoiding the need to deal with enumerations and attributes. Also the code related to the generation of SA code may be reused. In particular this is already being done by another tool developed in the context of the AVIACC project, which performs model checking of software using abstraction techniques.

Our preliminary results are satisfactory, and sufficient to illustrate the advantages of automatic verification. Even though not particularly optimized at the present stage, the tool seems to scale reasonably well - the array size in the example first shown in Figure 2.1 may well be increased up to thousands.

6.2 Future Work

Our current work on the development of the tool focuses on covering aspects of SPARK that are still not handled, including support for floating and fixed point types and properties related to them. Moreover, to make the tool fully automatic for some properties, we intend to create an instrumentation tool capable of inserting 'obvious' annotation related with over/underflow, array out of bounds accesses and division by zero also, importing SPARK contracts when possible.

Function inlining is another part of SPARK-BMC that needs improvement. It is expected that at the time of presenting this thesis such functionality is fully operational, however, at the time of writing it, only one

function at a time may be verified. If function calls exist, they must be inlined manually.

In the near future we want to check real applications and optimize the generation of Z3 expressions to reach scalability. We also intend to create encodings using unbounded integers, which may in certain circumstances be advantageous. Moreover, we expect to support different solvers in order to take advantage of the best from each one, possibly resorting to the why3 tool [BFMP11].

CBMC allows different loops to be unwound a different number of times. We also intend to have such a functionality and maybe go even further. For example, in CBMC, such a bound has to be given as an input parameter, specifying the loop's id and the bound. It requires that the user checks every loop id and writes it when invoking CBMC together with the bound. Whenever a loop is added, the loops' ids will change. To overcome this limitation, we intend to add a new annotation to be added immediately before each loop. With such an annotation, the user may specify the desired bound for each loop, and may also choose between an unwinding assertion or an unwinding assumption to be added after the unwound loop. If none is specified a default value will be used. Figure 6.1 presents an example of this approach. In this case, the loop would be unwound 10 times and an unwinding assumption would be used.

At a later stage we intend to extend the tool to handle concurrent programs, more precisely programs written in the RavenSPARK profile. The first step is to study the work presented in [LR09] and if possible implement it on SPARK-BMC.

A method we have never explored was to translate SPARK programs into an intermediate representation, like for example LLVM's IR^* and take advantage of existing BMCs of software like LLBMC. This approach is only possible if the SPARK code can be converted into LLVM's IR^*, but if this is possible only the implementation of this transformation is required. However, when the solver returns a counter-example, that counter-example must be propagated to the SPARK level. This is probably the most difficult task of this approach.

```
package body Factorial is
  function fact(X: Integer) return Integer
  is
    F, I: Integer;
  begin
    F := 1;
    I := 1;
    --% unwind(10, assumption);
    loop
      --% notOverflow(+,Integer,X,1);
      exit when ( I = X + 1 );
      --% notOverflow(*,Integer,F,I);
      F := F * I;
      --% notOverflow(+,Integer,I,1);
      I := I + 1;
    end loop;
    return F;
  end fact;
end Factorial;
```

Figure 6.1: Example of annotating loops

Bibliography

[ABDD07] Alex Aiken, Suhabe Bugrara, Isil Dillig, and Thomas Dillig. An overview of the Saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, New York, USA, 2007. ACM.

[Alt11] Altran Praxis. *SPARK - The SPADE Ada Kernel (including RavenSPARK)*, December 2011. Edition 7.2. Available from `http://docs.adacore.com/sparkdocs-docs/`.

[AMP09] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, January 2009.

[Bab08] Domagoj Babić. *Exploiting Structure for Scalable Software Verification.* PhD thesis, University of British Columbia, 2008.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc, Boston, USA, 2003.

[BB09] Robert Brummayer and Armin Biere. Boolector : An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, Inc., 2001.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th on Design Automation Conference*, pages 317–320, New York, USA, 1999. ACM.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franz, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver tool paper. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[BCM+92] Jerry Burch, Edmund Clarke, Kenneth Mcmillan, David Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Ber11] Stefan Berghofer. Verification of dependable software using SPARK and Isabelle. In *Proceedings of the 6th International Workshop on Systems Software Verification*, volume 24, Wadern, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[BH07] Domagoj Babić and AlanJ. Hu. Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 366–378, Berlin, Heidelberg, 2007. Springer-Verlag.

[BH08] Domagoj Babić and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, pages 211–220, New York, USA, 2008. ACM Press.

[BHMW09] Armin Biere, Marijn Heule, Hans Maaren, and Toby Walsh. *Handbook of Satisfiability.* IOS Press, 2009.

[BL05] Mike Barnett and K. Rustan Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 82–87, New York, USA, 2005. ACM.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 298–302, Berlin, Heidelberg, 2007. Springer-Verlag.

[CE81] Edmund Clarke and Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs*, pages 52–71, London, UK, 1981. Springer-Verlag.

[CES09] Edmund Clarke, Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, November 2009.

[CFMS12] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, July 2012.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, New York, USA, 1989. ACM.

[CGP99] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[Cha00] Roderick Chapman. Industrial experience with SPARK. *ACM SIGAda Ada Letters*, XX(4):64–68, December 2000.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer-Verlag.

[CKY03] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, pages 368–371, New York, USA, 2003. ACM.

[dCFP12] Daniela da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1264–1270, New York, USA, 2012. ACM Press.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[dMB11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–518, Berlin, Heidelberg, 2003. Springer-Verlag.

[FLL+02] Cormac Flanagan, K. Rustan Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, May 2002.

[GD07] Vijay Ganesh and David Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 519–531, Berlin, Heidelberg, 2007. Springer-Verlag.

[GGA05] Malay K Ganai, Aarti Gupta, and Pranav Ashar. DiVer : SAT-based model checking platform for verifying large scale systems. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 575–580, Berlin, Heidelberg, 2005. Springer-Verlag.

[GKSS08] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Chapter 2 satisfiability solvers. In *Handbook of Knowledge Representation*, pages 89 – 134. Elsevier Science, 2008.

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012.

[IBG+11] Franjo Ivančić, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Maeda Naoto, Hiroki Tokuoka, Takashi Imoto, and Miyazaki Yoshiaki. Dc2: A framework for scalable, scope-bounded software verification. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 133–142, Washington, USA, 2011. IEEE Computer Society.

[IYG+04] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. In *Preliminary proceedings of the 1th International Symposium on Leveraging Formal Methods*, pages 157–

164. Department of Computer Science, University of Cyprus, 2004.

[IYG+08] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, September 2008.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):1–54, October 2009.

[KC06] Daniel Kroening and Edmund Clarke. ANSI-C bounded model checker user manual. Technical Report 99, Carnegie Mellon University, Pittsburgh, PA 15213, 2006.

[LA04] Chris Lattner and Vikran Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–86, Washington, USA, 2004. IEEE Computer Society.

[LMFP13] Cláudio Belo Lourenço, Victor Cacciari Miraldo, Maria João Frade, and Jorge Sousa Pinto. SPARK-BMC : Checking SPARK code for bugs. In *Coletânea de Comunicações do 5º Simpósio de Informática in INForum 2013*, pages 242–253, 2013.

[LR09] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, August 2009.

[Mcm92] Kenneth Mcmillan. *Symbolic Mode Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[MFLP13] Victor Cacciari Miraldo, Maria João Frade, Cláudio Belo Lourenço, and Jorge Sousa Pinto. Experimenting with predicate

abstraction. In *Atas do 5º Simpósio de Informática in INForum 2013*, pages 102–114, 2013.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR*. In *Proceedings of the 4th International Conference on Verified Software: theories, tools, experiments*, pages 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1st edition, 1997.

[NMRW02] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2002. Springer-Verlag.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[SKW08] D'Vijay Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[XA05a] Yichen Xie and Alex Aiken. Saturn: A SAT-based tool for bug detection. In *Proceedings of the 17th International Conference on Computer Aided Verification*, pages 139–143, Berlin, Heidelberg, 2005. Springer-Verlag.

[XA05b] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. *ACM SIGPLAN Notices*, 40(1):351–363, January 2005.

[XA07] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, 29(3):16–58, May 2007.

[Xie06] Yichen Xie. *Static Detection of Software Errors.* PhD thesis, Standford University, 2006.

Appendix A

Installing SPARK-BMC

In this section we describe how to install and use SPARK-BMC. To be able to use the tool, one has first to install some required software. First of all, note that we do not provide any executable files. We do provide the Haskell source code and a Cabal package. Cabal is a system which provides an interface to describe the way in which libraries or programs are built. Having a Cabal package defined, it is possible to automatically compile programs using for that only one instruction. Since our tool is developed in Haskell a compiler for Haskell has also to be installed. We recommend the installation of 'The Haskell Platform'[1] which includes the ghc compiler (Glasgow Haskell Compiler) and Cabal.

Presently our tool only supports the Z3 solver for satisfiability checking. To install it, one should go to `http://z3.codeplex.com/` and download the installable for Windows or the source code for other platforms. The instructions to compile and install Z3 can be found on the web page.

Having the necessary tools installed it is time to download SPARK-BMC source code from the repository `https://bitbucket.org/vhaslab/spark-src`. A link can be found on the overview page to download it directly, or else to clone it with Mercurial[2].

[1] `http://www.haskell.org/platform/`
[2] `http://mercurial.selenic.com/`

Having the source code, the following operations must be performed:

```
cd spark-src
cabal configure
cabal install
```

If everything goes as expected, a binary file called `spark-bmc` should be created in the directory `.../spark-src/dist/build`.
To start using SPARK-BMC run:

```
./spark-bmc --help
```

to see the parameters that should be given as input.